МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ, КОМП'ЮТЕРНОЇ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Кафедра комп'ютеризованих систем управління

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри

_____Литвиненко О.Є.

«___» _____2020 р.

ДИПЛОМНА РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ "МАГІСТР"

Тема: «Система розпізнавання рукописного тексту»

Виконавець: Сирота С.В.

Керівник: Нечипорук В.В.

Нормоконтролер: Тупота Є.В.

Київ 2020

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної інженерії
Кафедра комп'ютеризованих систем управління
Освітнього ступеня магістр
Спеціальність 123 "Комп'ютерна інженерія"
(шифр, найменування)

Спеціалізація 123.02 "Системне програмування"
(шифр, найменування)

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____ Литвиненко О. Є.

«____»_____2020 р.

ЗАВДАННЯ
на виконання дипломної роботи (проекту)

Сироти Сергія Віталійовича
(прізвище, ім'я, по батькові випускника в родовому відмінку)

1. Тема роботи: Система розпізнавання рукописного тексту
затверджена наказом ректора від " 27 " серпня 2020 року № 1203 /ст.
2. Термін виконання роботи: з 05.10.2020 до 31.12.2020
3. Вихідні дані до роботи: Використання мови програмування *Python* для реалізації програмної системи розпізнавання рукописного тексту
4. Зміст пояснювальної записки (перелік питань, що підлягають розробці):
1) дослідження стану проблеми;
2) методи обробки рукописного тексту;
3) проектування модулю розпізнавання рукописного тексту
5. Перелік обов'язкового графічного матеріалу:
1) тришарова мережа для розпізнавання рукописних цифр;
2) збережені рисунки літери «А»;
3) вірогідності розпізнавання кожної літери;
4) вікно програми для розпізнавання рукописного тексту;
5) схема алгоритму розпізнавання рукописного тексту.

6. Календарний план

№ п/п	Етапи виконання дипломної роботи	Термін виконання етапів	Примітка
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			

7. Дата видачі завдання 05.10.2020

Керівник Нечипорук В.В.
(підпис)

Завдання прийняв до виконання Сирота С.В.
(підпис студента)

РЕФЕРАТ

Пояснювальна записка до дипломної роботи “Система розпізнавання рукописного тексту”: 90 с., 45 рис., 22 літературних джерела, 1 додаток.

РОЗПІЗНАВАННЯ ТЕКСТУ, НЕЙРОННА МЕРЕЖА, АНАЛІЗ РИСУНКІВ, ВИБІРКА, МАШИННЕ НАВЧАННЯ.

Мета дипломної роботи: аналіз методів розпізнавання рукописного тексту та їх реалізації у вигляді програмного забезпечення.

Об'єкт дослідження: розпізнавання рукописного тексту шляхом використання нейронної мережі.

Предмет система розпізнавання рукописного тексту.

Наукова значимість полягає у реалізації методу розпізнавання рукописного тексту на графічних зображеннях на основі використання нейронних мереж.

Практична значимість полягає у розробці програмного забезпечення, що планується використовувати у навчальному процесі, як практична складова у вивченні застосування нейронних мереж в системах розпізнавання графічних файлів.

Прогнозні припущення щодо подальшого розвитку матеріалів роботи полягає у встановленні та реалізації потенційних граничних значень точності розпізнавання рукописного тексту.

Публікації: Сирота С.В., Яковенко Л.В. Система розпізнавання рукописного тексту// Тези доповідей наук.-практ. конф. “Сучасні тенденції розвитку системного програмування” (25-26 листопада 2020 р.). – К.: НАУ, 2020. – С. 36.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ 6

ВСТУП 7

РОЗДІЛ 1 ДОСЛІДЖЕННЯ СТАНУ ПРОБЛЕМИ.....11

1.1. Аналіз використання перцептронів.....12

1.2. Аналіз використання сигмоподібних нейронів 19

1.3. Аналіз архітектури нейронних мереж.....24

1.4. Типові завдання розпізнавання 27

1.5. Висновки до розділу.....31

РОЗДІЛ 2 МЕТОДИ ОБРОБКИ РУКОПИСНОГО ТЕКСТУ 33

2.1. Проста мережа для класифікації рукописних цифр 33

2.2. Навчання з градієнтним спуском 38

2.3. Модифікація методу варіації градієнтного спуску.....47

2.4. Впровадження мережі для класифікації цифр.....50

2.5. Висновки до розділу.....62

РОЗДІЛ 3 ПРОЕКТУВАННЯ МОДУЛЮ РОЗПІЗНАВАННЯ РУКОПИСНОГО ТЕКСТУ..... 64

3.1. Описання програмного модуля розпізнавання рукописного тексту 64

3.2. Навчання нейронної мережі 71

3.3. Розпізнавання символів 79

3.4. Аналіз роботи основної частини 83

3.5. Тестування основних компонентів.....83

3.6. Робота з розпізнаванням 85

3.7. Висновки до розділу.....86

ВИСНОВКИ 87

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ..... 89

ДОДАТОК А 90

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

ВСТУП

Практично всі відомі і потрібні людині тексти сьогодні існують в цифровому вигляді: так до них простіше отримати доступ і використовувати для якихось завдань крім читання (наприклад, для швидкого пошуку інформації). З огляду на гігантський обсяг книг і періодичних видань, накопичених з часів винаходу друкарства, з цим завданням можна було впоратися тільки за допомогою автоматичних методів.

Тому ближче до кінця *XX* століття з'явилися і стали активно розвиватися технології оптичного розпізнавання символів (англ. *Optical character recognition*, скорочено – *OCR*). Найпростіший і до сих пір застосовується метод – матричний зіставлення: кожна буква в оригінальному документі розбивається на піксельні матриці, а потім зіставляється з матрицями, які є в комп'ютера. При збігу матриць буква вважається розпізнаної.

Щоб такий метод розпізнавання працював, для кожної рукописного тексту досить мати порівняно невелику кількість даних – зразків написання букв, за якими і робляться попіксельно матриці. Так, якщо у системи оптичного розпізнавання є матриці всіх 33 літер українського алфавіту з урахуванням можливих регістрів, а також всіх знаків, то вона зможе розпізнати будь-який україномовний текст.

Все, однак, не так просто. Цей метод контекстно-залежний і не може зробити крок в сторону від наявної у нього матриці: якщо пікселі в букві на оригінальному документі розташовані трохи інакше, то система в кращому випадку розпізнає її з помилкою (наприклад, замість п в тексті раптово з'явиться л), а в гіршому – не розпізнає зовсім.

Грубо кажучи, програма, яка вміє розпізнавати текст, набраний шрифтом *Times New Roman*, не розпізнає текст, набраний *Arial*, і навпаки.

Технології розпізнавання тексту, засновані виключно на матричному зіставленні, зараз досить універсальні і можуть працювати з усіма відомими

друкарськими шрифтами і всіма мовами (тому вони до сих пір широко використовуються).

При автоматичному розпізнаванні тексту може виникнути й інша складність: між розпізнаванням рукописного тексту прямо в процесі написання (свого роду режим «онлайн») і розпізнаванням вже готового тексту («офлайн») є велика різниця.

У першому випадку на допомогу приходять додаткові дані – фіксується процес введення: завдяки тому, що у комп'ютера є дані про ведення пера (тобто про те, як виводяться букви на екрані), він може розпізнати їх відразу ж.

Наприклад, рукописне про виводиться, починаючи майже з самої верхньої точки букви, а схожа петля у рукописного а починається з іншого місця, і навіть в тому випадку, якщо в слові «цікавіше» перші дві голосні розташувати дуже близько один до одного так, що вони зіллються і будуть нагадувати рукописне а з зайвої паличкою, проблеми не виникне: система, що стежила за процесом написання, зазначила, з якої точки перо початок виводити першу букву.

Крім того, під час письма трапляються паузи – це теж допоміжна і вельми корисна інформація. Так, якщо візьмемо слово «дихаєш», – відомий приклад того, наскільки нерозбірливим може бути український рукописний текст навіть для людини, – то комп'ютеру легше буде прочитати його, орієнтуючись на мікропаузи, що виникають між написанням окремих букв.

Хорошим матеріалом для порівняння двох методів розпізнавання почерків можуть послужити лікарські рецепти, які, як відомо, практично нечитабельні. Спостерігаючи за тим, як лікар виводить кожну окрему букву, швидше зрозуміємо, що він пише, ніж дивлячись на вже готові «каракулі».

З «офлайн» -розпізнавання, таким чином, все непросто – воно має справу з уже написаним текстом, без будь-якої інформації про те, як саме його писали. В цілому, програму для розпізнавання таких текстів навчають приблизно так само, як і у випадку з «онлайн» -розпізнаванням, але доводиться враховувати, що доступна інформація, необхідна для правильного впізнавання букв, тут дуже обмежена.

Проблема виникає тоді, коли програмі доводиться мати справу зі шрифтами, для яких достовірної піксельної матриці немає або її дуже важко створити, – йдеться, в першу чергу, про тексти, написані людиною від руки.

У випадку з рукописним текстом (або іншими рідкісними або нетиповими шрифтами) варіант зі звичайним зіставленням піксельних матриць може зовсім не працювати. В такому випадку використовується трохи інший спосіб – розпізнавання окремих образів (втім, воно теж оптичне).

Кожна буква, незважаючи на те, що пишеться різними людьми по-різному, все ж складається з однакових частин: в рукописної р – довга паличка, а в в – дві фігури, схожі на крапельки. У випадку з розпізнаванням окремих образів також використовується попіксельне порівняння, але варіантів того, як можуть виглядати літери, у системи набагато більше – просто тому, що там використовуються окремі їх частини.

У такому випадку кожен окремий знак – це вже вектор характерних для літери графічних ознак, а завдання зводиться до того, щоб знайти їх в початковому тексті: наприклад, за допомогою методу *k*-найближчих сусідів, використовуваного для вирішення завдань класифікації та кластеризації.

Робота подібних алгоритмів вимагає великої кількості розмічених даних, але сьогодні ця попередня робота в багатьох випадках вже пророблена. Наприклад, в датасеті *MNIST* міститься близько 70 тисяч зображень написаних від руки цифр, і точність розпізнавання у навчених на ньому алгоритмів дуже висока – для згортальних нейромереж вона становить понад 99 відсотків.

Алгоритми для розпізнавання рукописного тексту, зрозуміло, необхідні і зараз. Хоча б для того, наприклад, щоб розбирати ті ж лікарські рецепти. Але у програм *OCR* є завдання і іншого роду – дослідні.

У розпорядженні людства є безліч різноманітних рукописів – як порівняно нових, так і стародавніх, написаних на малозрозумілих і малознайомих мовах. Їх важко оцифрувати навіть за допомогою оператора-людини – адже часто треба не просто розпізнати кожен окремий символ або слово, необхідно відновити зміст тексту, колись очевидний для сучасників.

Труднощі часто виникають вже на рівні граматики – адже автор або переписувач цілком міг писати з помилками, і вони самі по собі представляють для істориків рукописного тексту інтерес, оскільки дозволяють «почути» давно замовкшу живу мову. Підключення словника або навіть корпусу для рукописного тексту розпізнавання не завжди дає результат, так як словник помилок не фіксує, а корпус може не знати саме цієї помилки.

Втім, розумне розпізнавання рукописного тексту здатне значно допомогти в роботі і з такими текстами.

Наприклад, в разі, якщо якість оцифрованого матеріалу не дуже гарна, класичне зіставлення матриць з опорою на словник, обмежена можливими для рукописного тексту словами, дозволяє виключити з тексту, що розпізнається свідомо невірні варіанти і навіть замінити їх вірними.

А якщо таку систему на великій кількості текстів навчити ще й граматиці розпізнаваного рукописного тексту (або, наприклад, використовувати *n*-грами, щоб обмежити варіанти слів, які в тексті можуть йти один за одним), то вона і зовсім зможе працювати практично без помилок.

Питання впровадження систем розпізнавання рукописних текстів для кириличних мов все ще вимагає доопрацювання, тому аналіз, проведений у даному дипломному дослідженні є актуальним на сьогоднішній день.

Мета дипломної роботи – аналіз методів розпізнавання рукописного тексту та їх реалізації у вигляді програмного забезпечення.

Об'єкт дослідження – розпізнавання рукописного тексту шляхом використання нейронної мережі.

Предмет дослідження – система розпізнавання рукописного тексту.

РОЗДІЛ 1
ДОСЛІДЖЕННЯ СТАНУ ПРОБЛЕМИ

Зорова система людини є одним із чудес світу. Розглянемо таку послідовність рукописних цифр (рис. 1.1).

Рис. 1.1. Приклад тексту прописом

Більшість людей без зусиль розпізнають ці цифри, як “504192”. Ця легкість оманлива. У кожній півкулі нашого мозку у людей є первинна зорова кора, також відома як V_1, що містить 140 мільйонів нейронів, з десятками мільярдів зв’язків між ними. І все-таки людський зір включає не просто V_1, а цілу серію зорових корок – V_2, V_3, V_4, V_5 – що робить поступово більш складну обробку зображень.

У голові людини суперкомп’ютер, налаштований еволюцією за сотні мільйонів років, і чудово пристосований для розуміння візуального світу. Розпізнати рукописні цифри непросто. Але майже вся ця робота робиться несвідомо. І тому людина зазвичай не розуміє, наскільки складну проблему вирішують зорові системи.

Труднощі візуального розпізнавання образів стають очевидними, якщо намагатися написати комп’ютерну програму для розпізнавання таких цифр, які наведені вище.

Нейронні мережі підходять до проблеми по-іншому. Ідея полягає в тому, щоб взяти велику кількість рукописних цифр, відомих як навчальні приклади (рис. 1.2), а потім розробити систему, яка може навчитися на цих прикладах навчання. Іншими словами, нейронна мережа використовує на прикладах автоматичне виведення правил розпізнавання рукописних цифр. Крім того, збільшуючи кількість навчальних прикладів, мережа може дізнатися більше про рукописний текст і, отже, підвищити його точність.

Рис. 1.2. Навчальна вибірка для нейронної мережі

Тож, хоча показано лише 100 навчальних цифр вище, є можливість створити кращий розпізнавач рукописного вводу, використовуючи тисячі, а то й мільйони чи мільярди прикладів навчання.

Найкращі комерційні нейромережі зараз настільки хороші, що їх використовують банки для обробки чеків і поштові відділення для розпізнавання адрес.

В роботі зосереджуємося на розпізнаванні рукописного вводу, оскільки це відмінна проблема прототипу для вивчення нейронних мереж загалом. Крім того, це чудовий спосіб розробити більш досконалі методи, такі як глибоке навчання.

1.1. Аналіз використання перцептронів

Перцептрон – це тип штучного нейрона. Перцептрони були розвинені у 1950–1960-х рр. вченим Френк Розенблат, натхненний раніше роботами від Уоррена Маккалока та Уолтера Піттса. Сьогодні частіше використовують інші моделі штучних нейронів – в багатьох сучасних роботах з нейронною мережею основною моделлю нейронів є така, що називається сигмоподібним нейроном.

Перцептрон приймає кілька двійкових входів x_1, x_2,… і видає єдиний двійковий вивід (рис. 1.3).

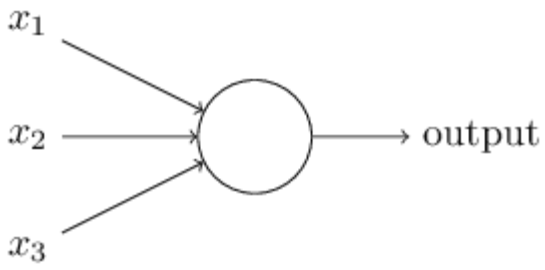

Рис. 1.3. Принцип роботи перцептрона

У наведеному прикладі перцептрон має три входи x_1, x_2, x_3. Загалом він може мати більше або менше вхідних даних. Розенблат запропонував просте правило для обчислення результату. Він ввів ваги $w_1, w_2, \ldots$, реальні числа, що виражають важливість відповідних входів для виходу.

Вихід нейрона 0 або 1, визначається тим, чи зважена сума $\sum_j w_j x_j$ менше або більше деякого порогового значення. Як і ваги, поріг – це дійсне число, яке є параметром нейрона. Якщо сказати точніше алгебраїчними термінами:

$$output = \begin{cases} 0, \text{ якщо } \sum_j w_j x_j \le \delta \\ 1, \text{ якщо } \sum_j w_j x_j > \delta \end{cases},$$

де δ – порогове значення.

Це основна математична модель. Перцептрон – це пристрій, який приймає рішення шляхом зважування доказів.

Наведемо не дуже реалістичний приклад, але його легко зрозуміти. Припустимо, наближаються вихідні, і особа, що приймає рішення (ОПР), чула, що у місті відбудеться фестиваль сиру. ОПР намагається вирішити, їхати на фестиваль чи ні. ОПР можете прийняти рішення, зваживши три фактори:

1) чи хороша погода?

2) хтось хоче супроводжувати ОПР?

3) фестиваль біля громадського транспорту (ОПР не володіє автомобілем).

Можемо представити ці три фактори за допомогою відповідних двійкових змінних x_1, x_2, x_3. Наприклад, $x_1 = 1$, якщо погода хороша, і $x_1 = 0$, якщо погода погана. Так само, $x_2 = 1$, якщо є компанія, і $x_2 = 0$, якщо ні, і так само знову для x_3 та громадського транспорту.

Тепер, припустимо, що ОПР обожнює сир, настільки, що із задоволенням поїде на фестиваль, навіть якщо відсутня компанія і до фестивалю важко дістатися. Але, можливо, ОПР ненавидить погану погоду, і ніяк не зможе піти на фестиваль, якщо погода погана. Можна використовувати перцептрони для моделювання такого типу прийняття рішень. Один із способів зробити це – вибрати вагу $w_1 = 6$ для погоди, $w_2 = 2$ і $w_3 = 2$ для інших умов.

Більше значення w_1 вказує на те, що погода для вас має велике значення, набагато більше, ніж наявність компанії або близькість громадського транспорту. Нарешті, припустимо, що вибрали поріг 5 для перцептрона. З цим вибором перцептрон реалізує бажану модель прийняття рішень, виводячи результати 1, коли погода хороша, і 0, коли погода погана. Немає різниці в результатах, чи хороша погода, є компанія або громадський транспорт поруч.

Змінюючи вагу та поріг, можна отримати різні моделі прийняття рішень. Наприклад, припустимо, замість 5 вибрали поріг 3. Тоді перцептрон вирішить, що слід їхати на фестиваль, коли погода хороша, або коли фестиваль знаходиться поруч із громадським транспортом, і буде наявна компанія. Іншими словами, це була б інша модель прийняття рішень. Зниження порогу означає, що більше готові прийняти позитивне рішення.

Очевидно, що перцептрон не є повною моделлю прийняття рішень людиною. Але приклад ілюструє те, як перцептрон може зважувати різні типи доказів для прийняття рішень. І повинно здатися правдоподібним, що складна мережа перцептронів може приймати досить тонкі рішення (рис. 1.4):

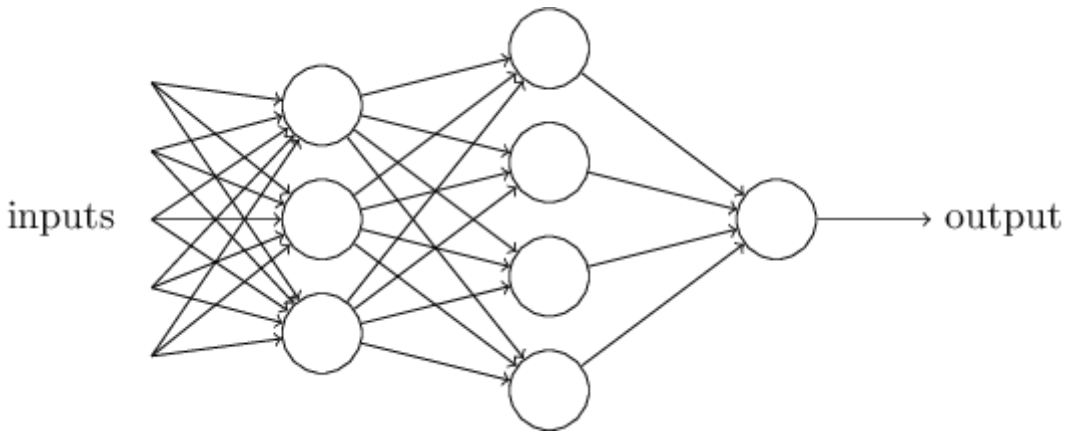

Рис. 1.4. Приклад роботи складного перцептрону

У цій мережі перша колонка перцептронів – те, що будемо називати першим шаром перцептронів – приймає три дуже простих рішення, зважуючи вхідні дані. У другому шарі кожен із цих перцептронів приймає рішення, зважуючи результати з першого рівня прийняття рішень. Таким чином, перцептрон у другому шарі може приймати рішення на більш складному і абстрактному рівні, ніж перцептрони в першому шарі. І ще більш складні рішення може приймати перцептрон у третьому шарі. Таким чином, багатошарова мережа перцептронів може брати участь у складних процесах прийняття рішень.

Коли визначали перцептрони, то обумовлювали, що перцептрон має лише один вихід. У мережі на рисунку 1.4 перцептрони виглядають так, ніби вони мають кілька виходів. Насправді вони все одно одиничні. Кілька стрілок виводу є лише корисним способом вказівки на те, що вихід перцептрона використовується як вхід для кількох інших перцептронів. Це менш громіздко, ніж малювати одну вихідну лінії, яка потім розпадається.

Давайте спростимо спосіб опису перцептронів. Стан $\sum_j w_j x_j > \delta$ це громіздко, і можемо внести дві нотаційні зміни, щоб спростити його.

Перша зміна – це писати $\sum_j w_j x_j$ як крапковий виріб:

$$w \cdot x \equiv \sum_j w_j x_j ,$$

де w і x є векторами, компонентами яких є ваги та входи відповідно.

Друга зміна полягає у переміщенні порогу на іншу сторону нерівності та заміщення його тим, що відоме як упередження перцептрона, $b \equiv -\delta$. Використовуючи зміщення замість порогового значення, правило перцептрону можна переписати:

$$output = \begin{cases} 0, \text{ якщо } w \cdot x + b \le 0 \\ 1, \text{ якщо } w \cdot x + b > 0 \end{cases}.$$

Можна сприймати зміщення як міру того, як легко отримати перцептрон для виведення 1. Або, кажучи більш біологічно, зміщення – це міра того, як легко змусити перцептрон спрацьовувати. Для перцептрона з дуже великим ухилом перцептрону надзвичайно легко вивести 1. Але якщо зміщення дуже негативне,

то перцептрону важко вивести 1. Очевидно, що введення упередженості – це лише невелика зміна в тому, як описуємо перцептрони.

Описання перцептрону як методу зважування доказів для прийняття рішень – це один з варінатів опису. Інший спосіб використання перцептронів – це обчислення елементарних логічних функцій, які зазвичай вважають основними обчисленнями, такі як *AND, OR* та *NAND*. Наприклад, припустимо, у нас є перцептрон з двома входами, кожен з вагою −2, і загальний упереджений – 3 (рис. 1.5). Ось наш перцептрон:

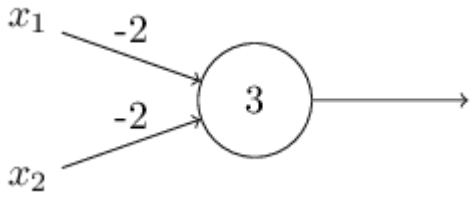

Рис. 1.5. Приклад перцептрона в нотації *NAND*

Тоді вхід 00 виробляє вихід 1, оскільки:

$$(-2)*0 + (-2)*0 + 3 = 3\ (-2)*0 + (-2)*0 + 3 = 3 > 0.$$

Ось, символ "∗" представлено, щоб зробити множення явними. Подібні розрахунки показують, що вхідні дані 01 і 10 виробляють вихід 1. Але вхідні дані 11 виробляють 0, оскільки:

$$(-2)*1 + (-2)*1 + 3 = -1\ (-2)*1 + (-2)*1 + 3 = -1 < 0.$$

Це приклад того, як перцептрон реалізує ворота *NAND*.

Приклад *NAND* показує, що можемо використовувати перцептрони для обчислення простих логічних функцій. Насправді можемо використовувати мережі перцептронів для обчислення будь-якої логічної функції взагалі. Причина в тому, що ворота *NAND* є універсальними для обчислень, тобто можемо побудувати будь-які обчислення за допомогою шлюзів *NAND*. Наприклад, можемо використовувати шлюзи *NAND* для побудови схеми, яка додає два біти, x_1 і x_2. Це вимагає обчислення побітової суми, $x_1 \oplus x_2$, а також біт перенесення,

для якого встановлено значення 1 коли обидва x_1 і x_2 є 1, тобто біт для перенесення – це лише побітовий добуток x_1x_2 (рис.1.6).

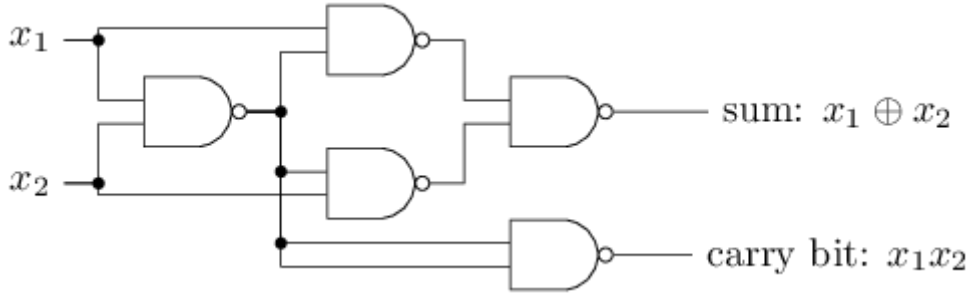

Рис. 1.6. Приклад реалізації воріт *NAND*

Щоб отримати еквівалентну мережу перцептронів, замінюємо всі ворота *NAND* на перцептрони з двома входами, кожен з вагою −2, і загальний упереджений 3. Ось отримана мережа. Зверніть увагу, що трохи переміщено перцептрон, що відповідає нижньому правому затвору *NAND,* лише для того, щоб полегшити малювання стрілок на схемі:

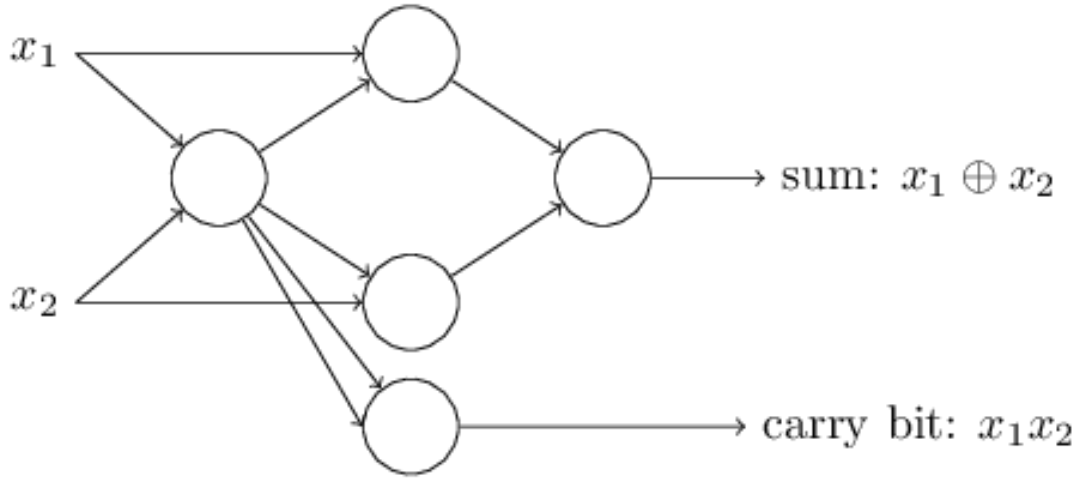

Рис. 1.7. Еквівалентна мережа перцептронів воріт *NAND*

Одним із помітних аспектів цієї мережі перцептронів є те, що вихід з самого лівого перцептрона використовується двічі як вхідний сигнал до самого нижнього перцептрона.

Коли визначали модель перцептрону, не вказували, чи дозволений такий вид подвійного виводу на те саме місце. Насправді це не має великого значення. Якщо не хочемо дозволити подібні речі, тоді необхідно об’єднати дві лінії в

єдине з'єднання з вагою -4 замість двох з'єднань з -2 вагами. З цією зміною мережа виглядає наступним чином, з усіма не маркованими вагами, рівними -2, всі упередження дорівнюють 3 і разова вага -4, як позначено на рисунку 1.8.

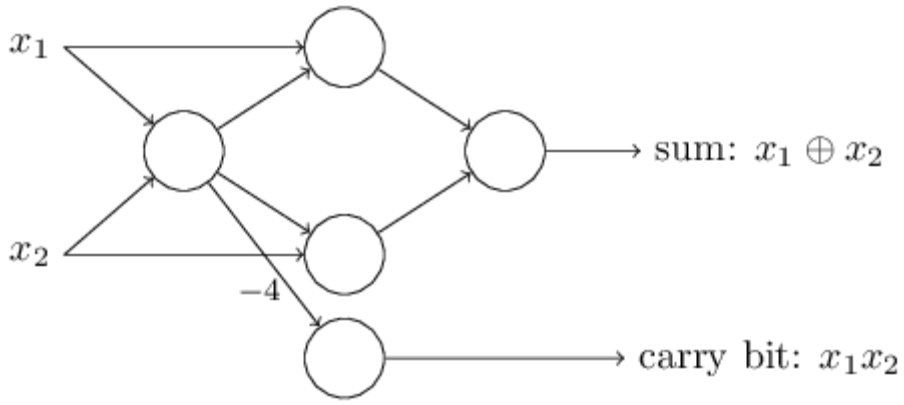

Рис. 1.8. Розрахунок в еквівалентній мережі перцептронів воріт *NAND*

До цього часу позначали введення типу x_1 і x_2 як змінні, що плавають ліворуч від мережі перцептронів. Насправді звичайно намалювати додатковий шар перцептронів – вхідний рівень – для кодування входів:

Це позначення для вхідних перцептронів, в яких маємо вихід, але не маємо входів (рис. 1.9),

Рис. 1.9. Представлення входів в еквівалентній мережі перцептронів воріт *NAND*

Це насправді не означає перцептрон без вхідних даних. Щоб побачити це, припустимо, у нас був перцептрон без входів. Потім зважена сума завжди була б нулем, і тому перцептрон виводив би 1 якщо $b > 0$, і 0 якщо $b \leq 0$. Тобто, перцептрон просто виводить фіксовану величину, а не бажане значення (x_1, у прикладі вище). Краще думати про вхідні перцептрони як про взагалі не про перцептрони, а про спеціальні одиниці, які просто визначаються для виведення бажаних значень, $x_1, x_2, \ldots x_n$.

Приклад суматора демонструє, як мережа перцептронів може бути використана для моделювання схеми, що містить безліч шлюзів *NAND*. І оскільки

ворота *NAND* універсальні для обчислень, з цього випливає, що перцептрони також універсальні для обчислень.

Обчислювальна універсальність перцептронів одночасно обнадіює і розчаровує. Це заспокоює, бо говорить нам, що мережі перцептронів можуть бути такими ж потужними, як будь-який інший обчислювальний пристрій. Але це також не відповідає дійсності, бо здається, ніби перцептрони – це просто новий тип воріт *NAND*.

Однак ситуація є іншою, ніж випливає з цього погляду. Можемо розробити алгоритми навчання, які можуть автоматично налаштовувати ваги та упередження мережі штучних нейронів. Це налаштування відбувається у відповідь на зовнішні подразники, без прямого втручання програміста. Ці алгоритми навчання дозволяють використовувати штучні нейрони способом, який кардинально відрізняється від звичайних логічних воріт. Замість того, щоб чітко викласти схему *NAND* та інших воріт, наші нейромережі можуть просто навчитися вирішувати проблеми, іноді проблеми, де надзвичайно складно безпосередньо спроектувати звичайну схему.

1.2. Аналіз використання сигмоподібних нейронів

Сишмоподібні нейрони передбачають навчання. Припустимо, у нас є мережа перцептронів, яку хотіли б використати, щоб навчитися вирішувати якусь проблему. Наприклад, вхідними даними в мережу можуть бути необроблені піксельні дані відсканованого рукописного зображення цифри. І хотіли б, щоб мережа вивчала ваги та упередження, щоб вихідні дані мережі правильно класифікували цифру. Щоб побачити, як може працювати навчання, припустимо, незначно змінимо деяку вагу (або упередженість) у мережі. Ця невелика зміна ваги спричиняє лише незначну відповідну зміну виходу з мережі. Як побачимо через мить, ця властивість зробить можливим навчання. Схематично це представлено на рисунку 1.10.

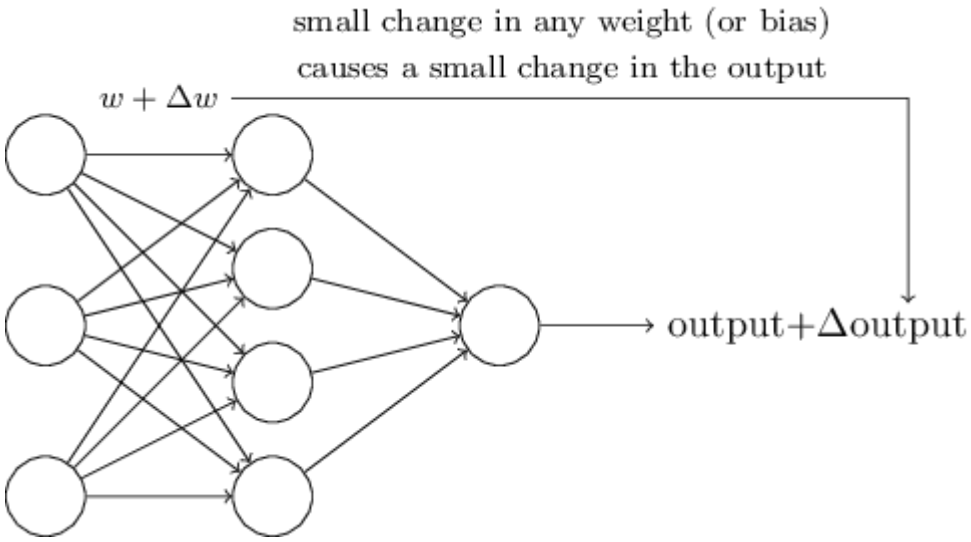

Рис. 1.10. Схематичне представлення навчання нейронів

Якби це було правдою, що невелика зміна ваги (або зміщення) спричиняє лише незначну зміну результату, тоді могли б використати цей факт для модифікації ваг та упереджень, щоб змусити нашу мережу вести себе більш належним чином. Наприклад, припустимо, що мережа помилково класифікувала зображення як "8", коли воно мало бути "9". Можна зрозуміти, як зробити невелику зміну ваги та упереджень, щоб мережа стала трохи ближчою до класифікації зображення як "9". І тоді повторювали б це, змінюючи ваги та упередження знову і знову, щоб отримати кращий і кращий результат. Мережа буде вчитися.

Проблема в тому, що це трапляється не так, коли наша мережа містить перцептрони. Насправді, невелика зміна ваги або зміщення будь-якого окремого перцептрона в мережі може іноді призвести до того, що вихід цього перцептрона повністю перевернеться, з 0 до 1. Потім це перевертання може призвести до того, що поведінка решти мережі повністю зміниться дуже складним чином. Отже, хоча ваш «9» тепер може бути класифікований правильно, поведінка мережі на всіх інших зображеннях, ймовірно, повністю змінилася якимось важко контрольованим способом. Це ускладнює розуміння того, як поступово змінювати ваги та упередження, щоб мережа наблизилася до бажаної поведінки. Можливо, є якийсь розумний спосіб обійти цю проблему. Але не одразу очевидно, як можемо отримати мережу перцептронів для навчання.

Можемо подолати цю проблему, запровадивши новий тип штучного нейрона, який називається сигмоподібний нейрон. Сигмоїдні нейрони схожі на перцептрони, але модифіковані таким чином, що незначні зміни їх ваги та упередженості спричиняють лише незначну зміну їх виходу. Це вирішальний факт, який дозволить мережі сигмоподібних нейронів вчитися.

Опишемо сигмоподібний нейрон. Зобразимо сигмоподібні нейрони так само, як зобразили перцептрони (рис. 1.11).

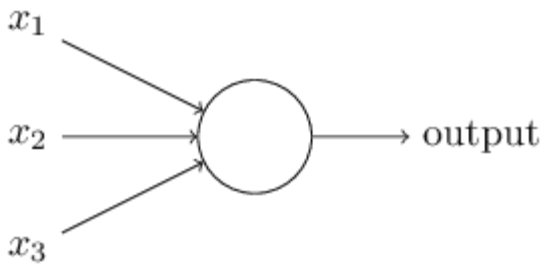

Рис. 1.11. Представлення сигмоподібного нейрону

Як і перцептрон, сигмоподібний нейрон має входи, x_1, x_2, x_3. Але замість того, щоб бути справедливим 0 або 1, ці входи також можуть приймати будь-які значення між 0 і 1. Так, наприклад, 0,638 – є допустимим входом для сигмоподібного нейрона. Також, як і перцептрон, сигмоподібний нейрон має ваги для кожного входу, w_1, w_2, і загальний упередженість, b. Але на виході немає 0 або 1. Натомість це:

$$\sigma\,(w \cdot x + b),$$

де σ – називається сигмоподібною функцією.

До речі, σ іноді називають логістичною функцією, а цей новий клас нейронів називають логістичними нейронами. Корисно пам’ятати цю термінологію, оскільки ці терміни використовуються багатьма людьми, що працюють з нейронними мережами. Однак дотримуватимемось сигмоподібної термінології, яка визначається:

$$\sigma(z) = \frac{1}{1+e^{-z}}.$$

Якщо сказати все трохи ясніше, вихід сигмоподібного нейрона з входами x_1, x_2, вагими w_1, w_2 та упередженнями b є:

$$\sigma(z) = \frac{1}{1 + e^{-\sum_j w_j x_j - b}}.$$

На перший погляд, сигмоподібні нейрони здаються дуже відмінними від перцептронів. Алгебраїчна форма сигмоїдної функції може здатися непрозорою і вразливою, якщо з нею ще не знайомі. Насправді між перцептронами та сигмоподібними нейронами є багато подібного, і алгебраїчна форма сигмоподібної функції виявляється більше технічною деталлю, аніж справжньою перешкодою для розуміння.

Припустимо, щоб зрозуміти схожість із моделлю перцептрону $z = w \cdot x + b$ – велике додатне число. Тоді $e^{-z} \approx 0$ і так $\sigma(z) \approx 1$.

Іншими словами, коли $z = w \cdot x + b$ велике і позитивне, вихід із сигмоподібного нейрону приблизно 1, як і для перцептрона.

Припустимо, з іншого боку, що $z = w \cdot x + b$ дуже негативний. Тоді $e^{-z} \to \infty$, і $\sigma(z) \approx 0$.

Тому, коли $z = w \cdot x + b$ дуже негативне, поведінка сигмоподібного нейрона також наближається до перцептрону. Це лише коли $w \cdot x + b$ має скромні розміри, що значно відхиляється від моделі перцептрону.

Точна форма σ не настільки важлива – важливе значення, яке має форма функції при побудові графіку. На рисунку 1.12 форма *sigmoid*-функції, а на рисунку 1.13 форма *step*-функції.

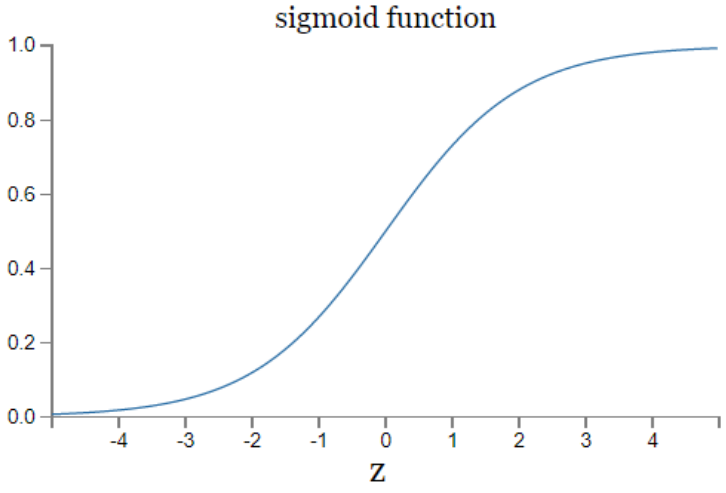

Рис. 1.12. Графічне представлення форми *sigmoid*-функції

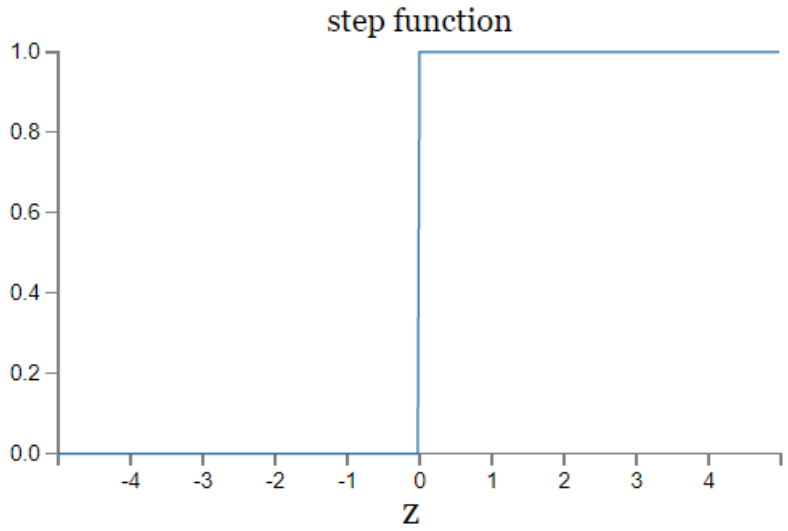

Рис. 1.13. Графічне представлення форми *sigmoid*-функції

Якщо σ насправді була б ступінчастою функцією, то сигмоподібний нейрон був би перцептроном, оскільки вихідний результат був би 1 або 0 залежно від того, чи $w \cdot x + b$ було позитивним чи негативним.

Власне, коли $w \cdot x + b = 0$ перцептрон виводить 0, тоді як функція кроку видає 1. Отже, строго кажучи, потрібно було б змінити функцію кроку в цій одній точці. За допомогою σ-функції отримуємо згладжений перцептрон. Дійсно, це плавність σ-функції є вирішальним фактом, а не її детальною формою. Плавність σ означає, що невеликі зміни Δw_j у вагах і Δb упередження призведуть до невеликих змін $\Delta output$ у виході з нейрона. Де сума перевищує всі ваги, w_j, і $\partial output / \partial w_j$ і $output / \partial b$ позначають часткові похідні від $output$ з вагою до w_j і b відповідно.

Хоча вираз вище виглядає складним, з усіма частковими похідними він насправді говорить про те, що $\Delta output$ є лінійною функцією змінх Δw_j і Δb у вагах та упередженості. Ця лінійність полегшує вибір незначних змін ваги та ухилів для досягнення будь-якої бажаної невеликої зміни у вихідній потужності. Отже, хоча сигмоподібні нейрони мають майже таку ж якісну поведінку, як і перцептрони, вони значно полегшують з'ясування того, як зміна ваги та упередженості змінить вихід.

Як саме слід інтерпретувати вихідні дані сигмоподібного нейрона. Очевидно, одна велика різниця між перцептронами та сигмоподібними нейронами полягає в тому, що сигмоподібні нейрони не просто виводять 0 або 1. Вони можуть мати як вихід будь-яке дійсне число між 0 і 1, тому такі значення, як 0,173… і 0,689… є корисними результатами. Вони можуть бути корисними, наприклад, якщо хочемо використовувати вихідне значення для представлення середньої інтенсивності пікселів на вводі зображення в нейронну мережу. Але іноді це може викликати неприємність. Припустимо, хочемо, щоб на виході з мережі вказувалося або «вхідне зображення – 9», або «вхідне зображення – не 9». Очевидно, що це було б найпростіше зробити, якби вихідний результат був 0 або 1, як у перцептроні. Але на практиці можемо створити конвенцію для вирішення цього питання, наприклад, вирішивши інтерпретувати будь-які результати, принаймні 0,5 як позначення "9", а будь-який вихід менше 0,5 як позначення "не 9". Необхідно завжди чітко заявляти, коли використовуємо таку конвенцію, тому це не повинно викликати плутанини.

1.3. Аналіз архітектури нейронних мереж

Покроково представимо нейронну мережу, яка може досить добре виконати класифікацію рукописних цифр. На першому кроці припустимо, що у нас є мережа (рис. 1.14).

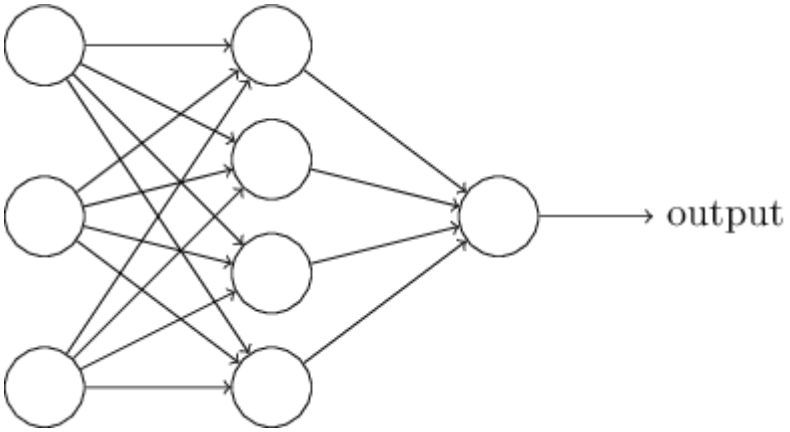

Рис. 1.14. Базова мережа розпізнавання рукописного тексту

Як згадувалося раніше, крайній лівий шар у цій мережі називається вхідним шаром, а нейрони всередині шару – вхідними нейронами. Крайній правий або вихідний шар містить вихідні нейрони або, як у цьому випадку, один вихідний нейрон. Середній шар називається прихованим шаром, оскільки нейрони в цьому шарі не є ні вхідними, ні вихідними. Наведена мережа має лише один прихований шар, але деякі мережі мають кілька прихованих шарів. Наприклад, наступна чотиришарова мережа має два приховані шари (рис. 1.15).

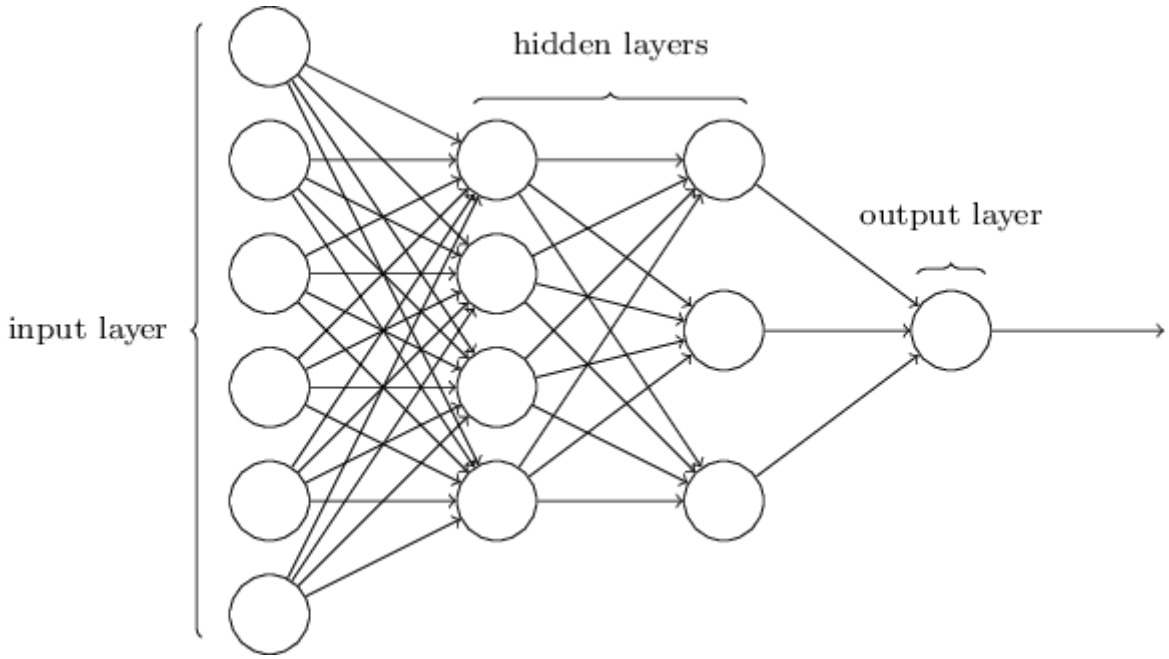

Рис. 1.15. Нейронна мережа з прихованим шаром

Дещо заплутано і з історичних причин такі багатошарові мережі іноді називають багатошаровими перцептронами або *MLP*, незважаючи на те, що вони складаються з сигмоподібних нейронів, а не перцептронів.

Дизайн вхідного та вихідного рівнів у мережі часто є простим. Наприклад, припустимо, що намагаємось визначити, чи зображено на рукописному зображенні «9» чи ні. Природним способом проектування мережі є кодування інтенсивності пікселів зображення у вхідні нейрони. Якщо зображення є 64 від 64 зображеннь у градаціях сірого, тоді б мали 4096 = 64 × 64 вхідні нейрони, інтенсивності яких масштабуються відповідно 0 і 1. Вихідний рівень міститиме лише один нейрон із вихідними значеннями менше ніж 0,5 що вказує "вхідне

зображення – це не 9", а значення більше 0,5 із зазначенням "вхідне зображення – 9".

Хоча дизайн вхідного та вихідного шарів нейронної мережі часто є простим, у дизайні прихованих шарів може бути ціле мистецтво. Зокрема, неможливо підвести підсумки процесу проектування прихованих шарів за допомогою кількох простих правил. Натомість дослідники нейронних мереж розробили безліч евристик проектування прихованих шарів, які допомагають людям вивести поведінку, яку вони хочуть, із своїх мереж. Наприклад, така евристика може бути використана, щоб допомогти визначити, як обміняти кількість прихованих шарів із часом, необхідним для навчання мережі.

До цього часу обговорювали нейромережі, де вихідні дані одного шару використовуються як вхідні дані для наступного шару. Такі мережі називаються нейронними мережами прямого зв'язку. Це означає, що в мережі немає циклів – інформація завжди подається вперед, ніколи назад. Якби у нас були цикли, опинилися б у ситуаціях, коли вхідні дані до σ функції залежали б від виходу. Це було б важко зрозуміти, і тому не потрідбно дозволяти робити такі дії.

Однак існують інші моделі штучних нейронних мереж, в яких можливі петлі зворотного зв'язку. Ці моделі називаються рекурентні нейронні мережі. Ідея цих моделей полягає в тому, щоб мати нейрони, які спрацьовують протягом певного обмеженого періоду часу, перш ніж стати в стан спокою. Цей вистріл може стимулювати інші нейрони, які можуть спалахнути трохи пізніше, також протягом обмеженого періоду. Це змушує ще більше нейронів спрацьовувати, і тому з часом отримуємо каскад нейронів, які стріляють. Цикли не викликають проблем у такій моделі, оскільки вихід нейрона впливає лише на якийсь пізній час, а не миттєво.

Повторювані нейронні мережі були менш впливовими, ніж мережі прямого зв'язку, частково тому, що алгоритми навчання для повторюваних мереж є (принаймні на сьогоднішній день) менш потужними. Але періодичні мережі все ще надзвичайно цікаві. Вони набагато ближчі суттю до того, як працює наш мозок, ніж мережі споживачів. І цілком можливо, що періодичні мережі можуть вирішити важливі проблеми, які можуть бути вирішені лише з великими

труднощами мережами прямого зв'язку. Однак, щоб обмежити наш обсяг, у даному розділі зосередимось на більш широко використовуваних мережах прямого зв'язку.

1.4. Типові завдання розпізнавання

Хоча нейронна мережа забезпечує вражаючу продуктивність, ця продуктивність є дещо загадковою. Ваги та упередження в мережі були виявлені автоматично. А це означає, що не відразу маємо пояснення того, як мережа робить те, що робить. Для цього необхідно знайти якийсь спосіб зрозуміти принципи, за якими наша мережа класифікує рукописні цифри. І, враховуючи такі принципи, знайти варіанти покращення.

Щоб поставити ці питання більш суворо, припустимо, що через кілька десятиліть нейронні мережі ведуть до штучного інтелекту (ШІ). У перші дні досліджень ШІ люди сподівались, що зусилля з побудови ШІ також допоможуть зрозуміти принципи інтелекту та, можливо, функціонування людського мозку. Але, можливо, результатом буде те, що в підсумку не зрозуміємо ні мозок, ні те, як працює штучний інтелект.

Щоб вирішити ці питання, давайте згадаємо інтерпретацію штучних нейронів, як засобу зважування доказів. Припустимо, хочемо визначити, чи зображено зображення людського обличчя чи ні (рис. 1.16). Можна було б атакувати цю проблему так само, як атакували розпізнавання рукописного вводу – використовуючи пікселі на зображенні як вхід до нейронної мережі, при цьому на виході з мережі один нейрон вказує або "Так, це обличчя", або "Ні, це не обличчя ".

Припустимо, робимо це, але не використовуємо алгоритм навчання. Натомість спробуємо спроектувати мережу вручну, вибираючи відповідні ваги та ухили.

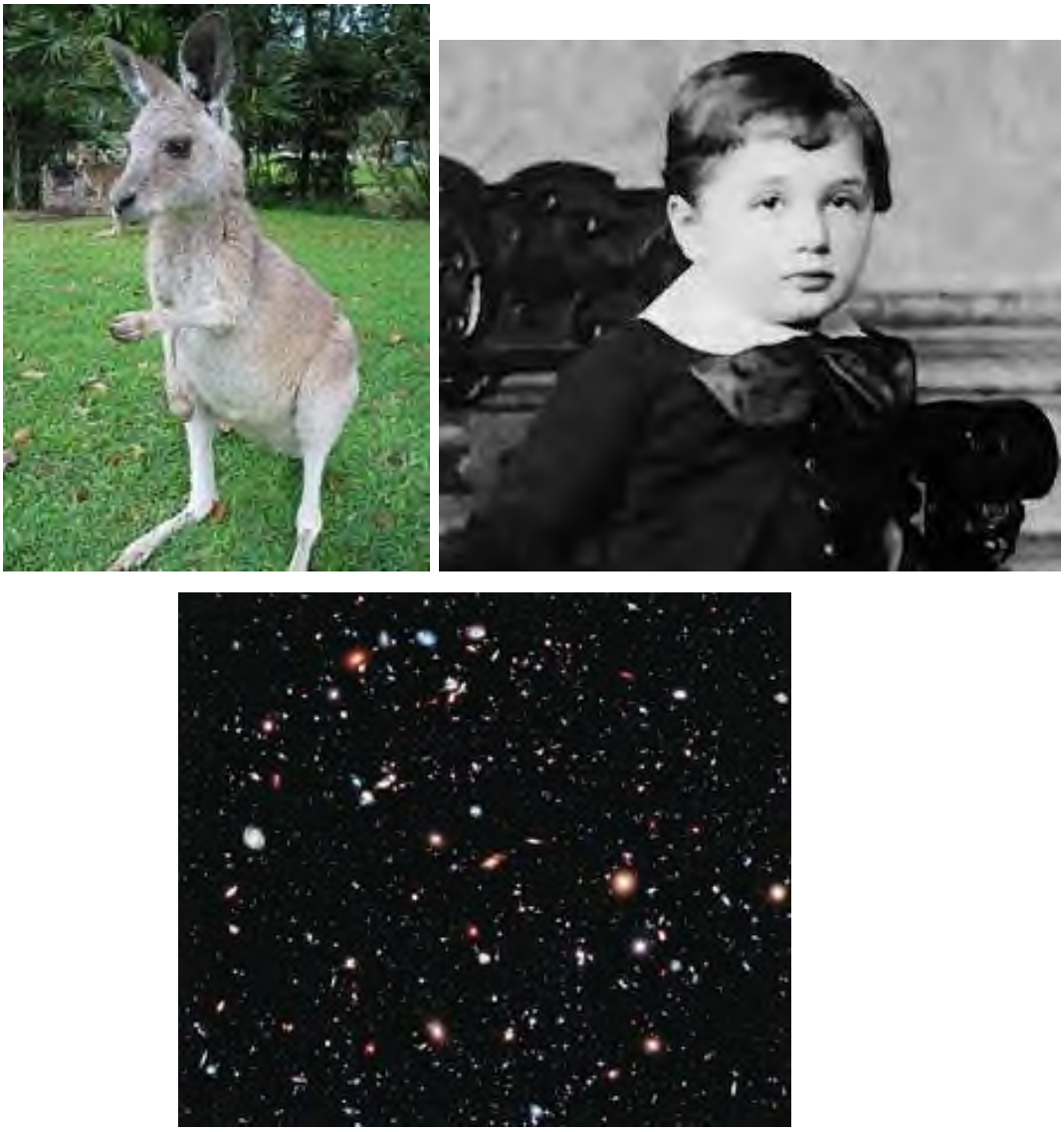

Рис. 1.16. Група зображень для задачі пошуку людського обличчя

Але для вирішення даної задачі без використання нейронних мереж є можливість застосувати евристику, за рахунок розкладання проблеми на підзадачі: «чи має зображення око вгорі ліворуч?», «чи має він око вгорі праворуч?», «у нього є ніс посередині?», «у нього є рот внизу посередині?, «чи є волосся зверху?» і т.д.

Якщо відповіді на кілька з цих запитань – "так", а то й просто "ймовірно, так", тоді зробимо висновок, що зображення, швидше за все, буде обличчям. І навпаки, якщо відповіді на більшість запитань – "ні", то зображення, ймовірно, не є обличчям.

Звичайно, це лише груба евристика, і вона страждає від багатьох недоліків. Можливо, людина лиса, тому у неї немає волосся. Можливо, можемо бачити

лише частину обличчя, або обличчя знаходиться під кутом, тому деякі риси обличчя затемнені. Тим не менше, евристика передбачає, що якщо можемо вирішити підзадачі за допомогою нейронних мереж, то, можливо, можемо побудувати нейронну мережу для виявлення обличчя, об'єднавши мережі для підпроблем. Ось можлива архітектура з прямокутниками, що позначають підмережі. Зверніть увагу, що це не задумано як реалістичний підхід до вирішення проблеми виявлення обличчя, швидше, це допоможе побудувати інтуїцію про те, як функціонують мережі. На рисунку 1.17 зображено архітектуру нейронної мережі для вирішення задачі пошку зображень з людським обличчям.

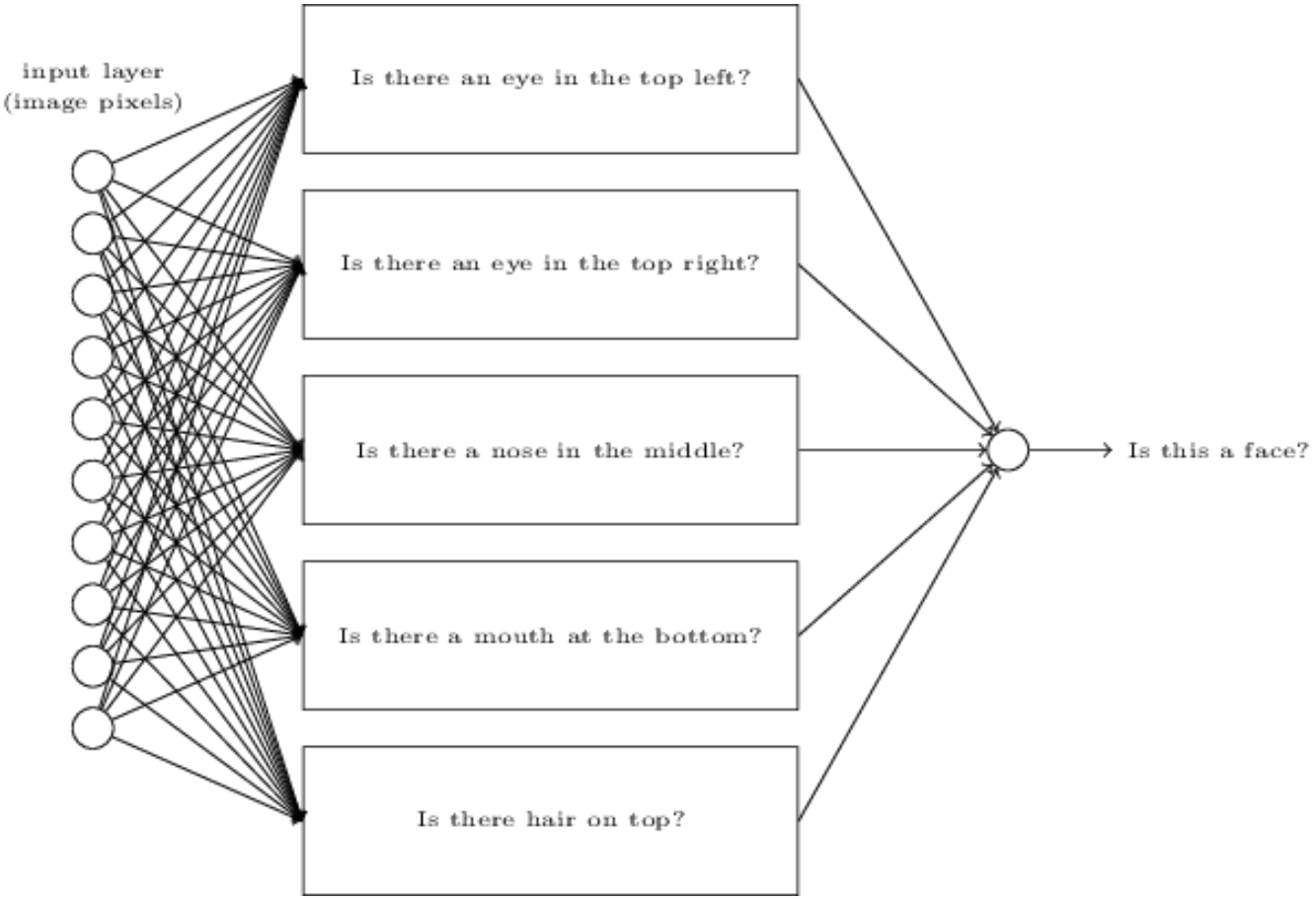

Рис. 1.17. Архітектура нейронної мережі для вирішення задачі пошку зображень з людським обличчям

Також ймовірно, що підмережі можуть бути розкладені. Припустимо, розглядаємо питання: "Чи є око вгорі ліворуч?". Це можна розкласти на такі питання, як: "Чи є брова?"; «Чи є вії?»; "Чи є райдужка?" і так далі. Звичайно, ці запитання повинні справді включати також позиційну інформацію – "Чи є брова

вгорі ліворуч і над райдужкою?" і т.п. питання, – але для спрощення не будемо збільшувати кількість питань, тоді мережа, щоб відповісти на запитання "Чи є око вгорі ліворуч?" тепер можна розкласти (рис. 1.18).

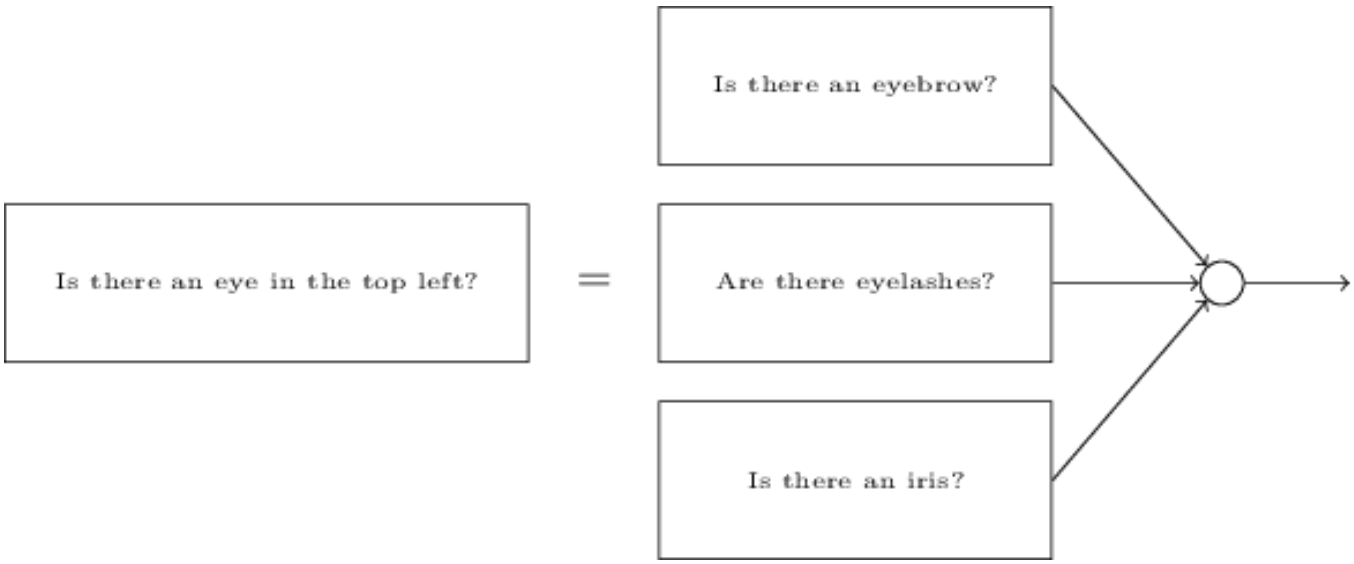

Рис. 1.18. Розкладання нейронної мережі на шари

Ці питання теж можна розбити, все далі і далі, за допомогою декількох шарів. Зрештою, будемо працювати з підмережами, які відповідають на запитання настільки прості, що на них можна легко відповісти на рівні одиничних пікселів. Ці питання можуть, наприклад, стосуватися наявності чи відсутності дуже простих фігур у певних точках зображення. На такі запитання можуть відповісти поодинокі нейрони, підключені до необроблених пікселів на зображенні.

Кінцевим результатом є мережа, яка розбиває дуже складне запитання – показує це зображення обличчя чи ні – на дуже прості запитання, що відповідають на рівні одиничних пікселів. Він робить це через безліч шарів, причому ранні шари відповідають на дуже прості та конкретні запитання щодо вхідного зображення, а пізніші шари вибудовують ієрархію все більш складних та абстрактних концепцій. Мережі з таким різновидом багатошарової структури – два або більше прихованих шарів – називаються глибокими нейронними мережами.

Не має сенсу самостійно розробляти ваги та упередження в мережі. Натомість бажано використовувати алгоритми навчання, щоб мережа могла

автоматично вивчати ваги та упередження – і, отже, ієрархію концепцій – з навчальних даних. Дослідники у 1980-х і 1990-х роках намагалися використовувати стохастичний градієнтний спуск і зворотнє поширення для підготовки глибоких мереж. На жаль, за винятком декількох спеціальних архітектур, їм не сильно пощастило. Мережі вчились, але дуже повільно, а на практиці часто занадто повільно, щоб бути корисними.

З 2006 року було розроблено набір методик, які дозволяють навчатись у глибоких нейронних мережах. Ці методи глибокого навчання засновані на стохастичному градієнтному спуску та зворотньому розповсюдженні, але також вводять нові ідеї. Ці методи дозволили навчити набагато глибші (і більші) мережі – тепер люди регулярно тренують мережі з 5-10 прихованими рівнями. І, виявляється, вони працюють набагато краще для багатьох проблем, ніж неглибокі нейронні мережі, тобто мережі з одним єдиним прихованим шаром. Причиною, звичайно, є здатність глибоких мереж будувати складну ієрархію понять. Це трохи схоже на те, як звичайні рукописного тексту програмування використовують модульний дизайн та ідеї про абстракцію для створення складних комп’ютерних програм. Порівняння глибокої мережі з неглибокою мережею трохи схоже на порівняння рукописного тексту програмування з можливістю здійснювати функціональні виклики до розірваного рукописного тексту без можливості здійснення таких дзвінків. Абстракція набуває в нейронних мережах іншої форми, ніж у звичайному програмуванні, але це так само важливо.

1.5. Висновки до розділу

В розділі розглянуто основні питання перспективності використання нейронних мереж для вирішення задачі розпізнавання рукописного тектсу.

Було визначено, що використання сучасних високоінтелектуальних інформаційних комп’ютерних технологій у сфері людської діяльності вимагає кардинальної зміни в управлінні автоматизованими системами для більш зручного та раціонального їх використання. До основних галузей систем

розпізнавання рукописного тексту можна віднести оцифровування рукописних текстів.

Окремо було розглянуто основні підходи до розуміння нейронних мереж і різниці між перцептронами та нейронними мережами з сигмоподібними функціями.

РОЗДІЛ 2
МЕТОДИ ОБРОБКИ РУКОПИСНОГО ТЕКСТУ

На основі відомостей щодо побудови нейронної мережі, які було представлено в розділі 1, побудуємо власну нейронну мережу для розв'язку задачі розпізнавання рукописного тексту.

2.1. Проста мережа для класифікації рукописних цифр

Можемо розділити проблему розпізнавання рукописних цифр на дві підзадачі:

1) розбити зображення, що містить багато цифр, на послідовність окремих зображень, кожна з яких містить одну цифру (рис. 2.1);

2) розпізнавання окремих цифр.

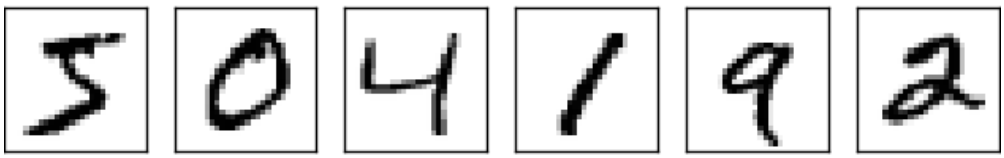

Рис. 2.1. Результат розбиття зображення

Люди вирішують цю проблему сегментації з легкістю, але комп'ютерній програмі важко правильно розбити зображення. Після сегментації зображення програмі потрібно класифікувати кожну окрему цифру. Отже, наприклад, хотіли б, щоб наша програма визнала, що перша цифра 5.

Зосередимося на написанні програми для вирішення другої проблеми, тобто класифікації окремих цифр. Робимо це, оскільки виявляється, що проблему сегментації вирішити не так складно, як тільки у вас є хороший спосіб класифікації окремих цифр. Існує багато підходів до вирішення проблеми сегментації. Одним із підходів є випробування багатьох різних способів сегментування зображення за допомогою індивідуального класифікатора цифр

для оцінки кожної пробної сегментації. Пробна сегментація отримує високий бал, якщо окремий класифікатор цифр впевнений у своїй класифікації у всіх сегментах, і низький бал, якщо класифікатор має багато проблем в одному або декількох сегментах. Ідея полягає в тому, що якщо десь у класифікатора виникають проблеми, то це, випливає з того, що сегментація була вибрана неправильно. Ця ідея та інші варіації можуть бути використані для вирішення проблеми сегментації досить добре. Тому замість того, щоб турбуватися про сегментацію, зосередимось на розробці нейронної мережі, яка може вирішити проблему, яка є більш цікавою та складною, а саме розпізнавання окремих рукописних цифр.

Для розпізнавання окремих цифр будемо використовувати тришарову нейронну мережу (рис. 2.2)

Рис. 2.2. Тришарова мережа для розпізнавання рукописних цифр

Вхідний рівень мережі містить нейрони, що кодують значення вхідних пікселів. Як обговорювалося в наступному розділі, наші навчальні дані для мережі складатимуться з багатьох 28 на 28 піксельні зображення відсканованих рукописних цифр, і тому вхідний шар містить 784 = 28 × 28 нейрони. Для простоти не висвітлюємо більшість 784 вхідних нейронів на схемі вище. Вхідні пікселі мають шкалу сірого, значенням 0,0 що представляє білий, значення 1.0 представляє чорний, а між значеннями представляє поступово темніючі відтінки сірого.

Другий рівень мережі – це прихований шар. Позначаємо кількість нейронів у цьому прихованому шарі і будемо експериментувати з різними значеннями. Наведений приклад ілюструє невеликий прихований шар, що містить просто n = 15 нейронів.

Вихідний рівень мережі містить 10 нейронів. Якщо перший нейрон спрацьовує, тобто має $output \approx 1$, це вказуватиме на те, що мережа вважає цифру 0. Якщо другий нейрон спрацьовує, то це вкаже на те, що мережа вважає цифру 1. І так далі. Трохи точніше, пронумеруємо вихідні нейрони 0 через 9, і з'ясуємо, який нейрон має найвище значення активації. Якщо цей нейрон, скажімо, номер 66, тоді наша мережа здогадається, що вхідною цифрою було 66. І так далі для інших вихідних нейронів.

Використовуємо 10 вихідних нейронів, тому що необхідно визначити, яка цифра (0, 1, 2, …, 9) відповідає вхідному зображенню. Начебто природним способом цього є використання справедливого 4 вихідного нейрону, розглядаючи кожен нейрон, який набуває двійкове значення, залежно від того, чи є вихід нейрона ближчим до 0 або до 1. Для кодування відповіді достатньо чотирьох нейронів, оскільки $2^4 = 16$ більше 10 можливих значень для вхідної цифри.

Кінцеве виправдання є емпіричним: можемо випробувати обидва проекти мережі, і виявляється, що для цієї конкретної проблеми мережа з 10 вихідними нейронами вчаться розпізнавати цифри краще, ніж мережа на 4 вихідні нейрони.

Щоб зрозуміти, для чого це робимо, корисно думати про те, що робить нейронна мережа з перших принципів. Розглянемо спочатку випадок, коли використовуємо 10 вихідних нейронів. Зупинимось на першому вихідному

нейроні, тому, який намагається вирішити, чи є цифра 0. Це робиться шляхом зважування доказів із прихованого шару нейронів. Проаналізуємо, що роблять ці приховані нейрони. Припустимо, що перший нейрон у прихованому шарі визначає, чи є зображення, таким як показано на рисунку 2.3.

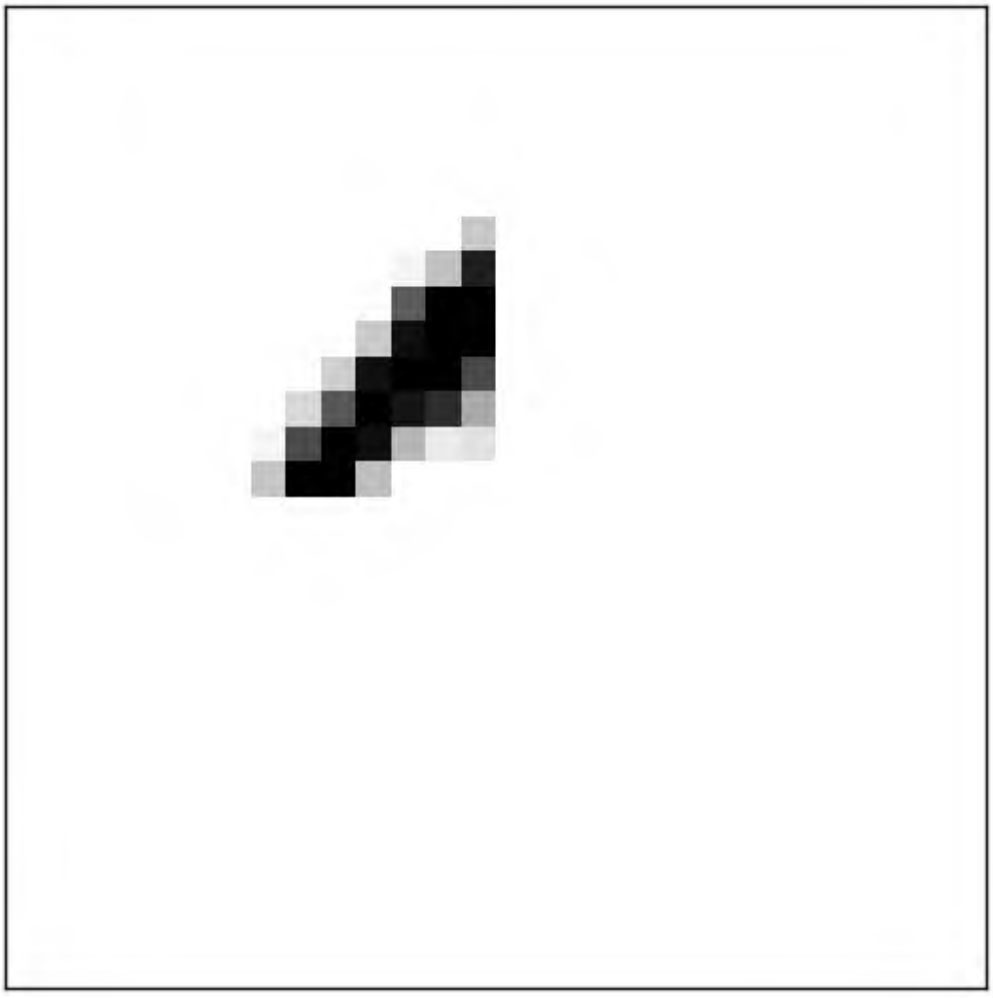

Рис. 2.3. Приклад покрокового розв’язку задачі розпізнавання

Це можна зробити, сильно взваживши вхідні пікселі, які перекриваються із зображенням, і лише злегка зваживши інші входи. Подібним чином припустимо, що другий, третій і четвертий нейрони в прихованому шарі визначають, чи є зображення з рисунку 2.4.

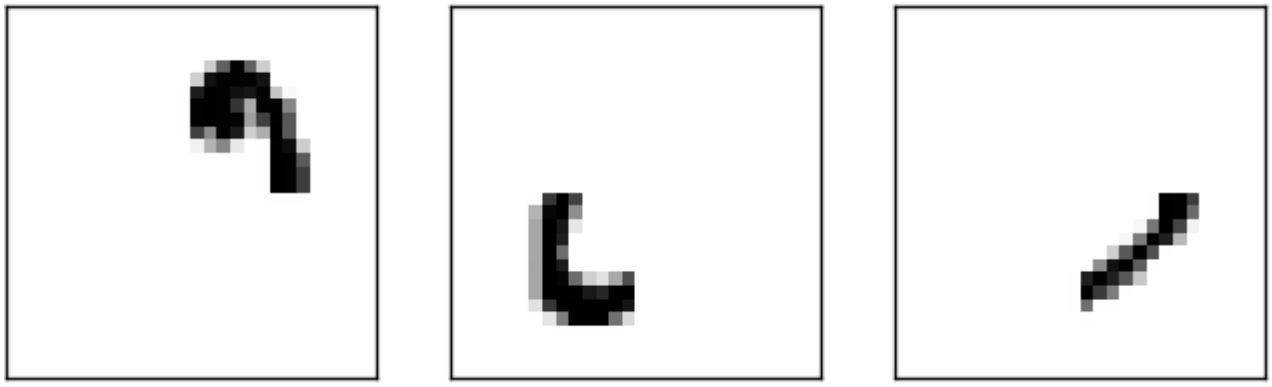

Рис. 2.4. Приклади зображень для 2, 3 та 4-го нейронів

Ці чотири зображення разом складають зображення 0, яке представлено у рядку цифр, що показано раніше (рис. 2.5).

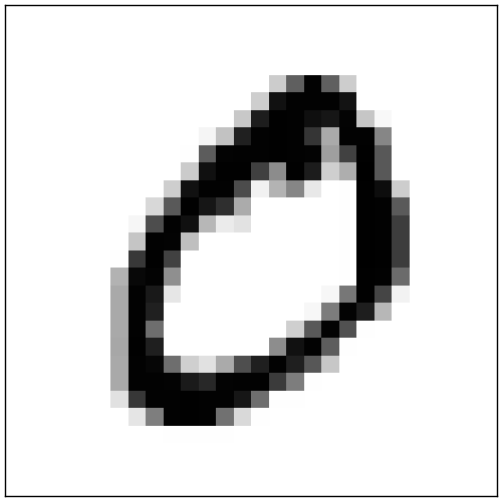

Рис. 2.5. Зображення рукописного “0”

Отже, якщо всі чотири з цих прихованих нейронів спрацьовують, то можемо зробити висновок, що цифра – це 0. Звичайно, це не єдиний доказ, за яким можемо зробити висновок, що зображення було 0 – могли б законно отримати 0 багатьма іншими способами (скажімо, через переклади вищевказаних зображень або незначні спотворення). Але здається безпечним стверджувати, що принаймні в цьому випадку зробимо висновок, що введеним значенням було 0.

Припустивши, що нейронна мережа функціонує таким чином, можемо дати правдоподібне пояснення, чому це краще мати 10 виходів *sd* з мережі, а не 4. Якби мали 4 виходи, тоді перший вихідний нейрон намагався б вирішити, яким був найзначніший біт цифри. І немає простого способу пов’язати цей

найважливіший біт із простими фігурами, як показано вище. Важко уявити, що існує якась вагома історична причина, через яку форми компонента цифри будуть тісно пов'язані (скажімо) з найважливішим бітом у вихідному коді.

Тепер, з усім сказаним, це все лише евристика. Ніщо не говорить про те, що тришарова нейронна мережа повинна працювати так, як описано, приховані нейрони виявляють прості форми компонентів. Можливо, розумний алгоритм навчання знайде якесь призначення ваг, яке дозволяє лише використовувати 44 вихідні нейрони. Але як евристичний спосіб мислення, який було описано, працює досить добре, і може заощадити вам багато часу при розробці хороших архітектур нейронних мереж.

2.2. Навчання з градієнтним спуском

Тепер, коли у нас є дизайн нашої нейронної мережі, визначимо як вона може навчитися розпізнавати цифри. Перше, що знадобиться, це набір даних, на якому можна вчитися – так званий навчальний набір даних. Будемо використовувати набір даних *MNIST*, який містить десятки тисяч відсканованих зображень рукописних цифр, а також їх правильну класифікацію. Назва *MNIST* походить від того, що це модифікована підмножина двох наборів даних, зібраних *NIST* (Національний інститут стандартів і технологій) США.

Як бачимо, ці цифри насправді такі ж, як і наведені на на початку цього розділу. Звичайно, під час тестування нашої мережі попросимо її розпізнати зображення, яких немає в навчальному наборі.

Дані *MNIST* складаються з двох частин. Перша частина містить 60 000 зображень, які будуть використані як навчальні дані. Ці зображення – це відскановані зразки почерку від 250 осіб, половина з яких – співробітники Бюро перепису населення США, а половина – учні середніх шкіл. Зображення мають градації сірого та мають розмір 28 на 28 пікселів. Друга частина набору даних *MNIST* – це 10 000 зображень, які будуть використані як тестові дані. Знову ж таки, це 28 на 28 зображень у градаціях сірого. Використаємо дані тесту, щоб оцінити, наскільки добре наша нейронна мережа навчилася розпізнавати цифри.

Щоб зробити це хорошим тестом успішності, дані тесту були взяті у 250 осіб, відмінних від вихідних даних про навчання (хоча все-таки група розділена між працівниками Бюро перепису населення та студентами середніх шкіл). Це допомагає упевнитись, що система може розпізнавати цифри від людей, які б не писали.

Будемо використовувати позначення x для позначення навчального матеріалу. Буде зручно розглядати кожен навчальний матеріал x як $28 \times 28 = 784$ - вимірний вектор. Кожен запис у векторі представляє значення сірого для одного пікселя на зображенні. Позначимо відповідний бажаний результат через $y = y(x)$, де y є 10-вимірний вектор. Наприклад, якщо певний навчальний образ, x, зображує 6, тоді $y(x) = (0,0,0,0,0,0,1,0,0,0)^T$ – бажаний вихід з мережі. Зауважте, що T – операція транспонування, перетворення вектора рядка у звичайний (стовпець) вектор.

Також необхідний алгоритм, який дозволяє знаходити ваги та упередження, щоб вихідні дані з мережі наближались $y(x)$ для всіх навчальних матеріалів x. Щоб кількісно визначити, наскільки добре досягаємо цієї мети, визначаємо функцію витрат. Іноді називають втратою або цільовою функцією. У даній роботі використовуємо термін функція витрат, але вам слід звернути увагу на іншу термінологію, оскільки вона часто використовується в наукових роботах та інших обговореннях нейронних мереж.

Ось, w позначає сукупність усіх ваг в мережі, b всі упередження, n – загальна кількість навчальних матеріалів, a – вектор виходів з мережі, коли x є вхідними даними, а сума перевищує всі вхідні дані, x. Звичайно, вихід a залежить від x, w і b, але щоб спростити позначення, чітко не вказують на цю залежність. Позначення $\|V\|$ просто позначає звичайну функцію довжини для вектора v. Викликаємо C функція квадратних витрат, її також іноді називають середньоквадратичною помилкою або просто *MSE*. Перевіряючи форму функції квадратних витрат, бачимо це $C(w, b)$ є невід'ємним, оскільки кожен доданок у сумі невід'ємний. Крім того, вартість $C(w, b)$ стає малою, тобто $C(w, b) \approx 0$, саме коли $y(x)$ приблизно дорівнює виходу a, для всіх навчальних матеріалів x. Отже,

наш алгоритм навчання добре зробив роботу, якщо він може знаходити ваги та упередження $C(w, b) \approx 0$. Навпаки, було б не так добре, коли $C(w, b)$ великий – це означало б, що $y(x)$ не наближається до результату a для великої кількості входів. Тож метою нашого навчального алгоритму буде мінімізація витрат $C(w, b)$ як функція ваг та упереджень. Іншими словами, хочемо знайти набір ваг та упереджень, які роблять вартість якомога меншою. Зробимо це за допомогою алгоритму, відомого як градієнтний спуск.

При розв'язку цікавить перш за все кількість зображень, правильно класифікованих мережею. Можна спробувати максимізувати це число безпосередньо, а не мінімізувати проксі-міру, як квадратичну вартість. Проблема в тому, що кількість правильно класифікованих зображень не є гладкою функцією ваг та упереджень у мережі. Здебільшого, невеликі зміни ваги та упереджень не спричинять жодних змін у кількості тренувальних зображень, правильно класифікованих. Це ускладнює з'ясування того, як змінити ваги та упередження, щоб отримати кращі показники. Якщо замість цього використовуємо функцію плавної вартості, як квадратичну вартість, виявляється в якій легко зрозуміти, як зробити невеликі зміни у вагах та упередженнях, щоб отримати покращення у вартості.

Навіть з огляду на те, що хочемо використовувати функцію плавної вартості, все одно можна задатися питанням, чому обираємо квадратичну функцію, яка використовується в рівнянні. Це не досить спеціальний вибір. Можливо, якби вибрали іншу функцію витрат, отримали б зовсім інший набір мінімізації ваг та упереджень. Це поважна проблема, і пізніше знову переглянемо функцію витрат і внесемо деякі зміни. Однак квадратична функція витрат рівняння чудово працює для розуміння основ навчання в нейронних мережах, тому наразі цього дотримуватимемось.

Підбиваючи підсумки, наша мета у навчанні нейронної мережі – знайти ваги та упередження, які мінімізують функцію квадратних витрат $C(w, b)$. Це добре поставлена проблема, але вона має багато відволікаючої структури, яку поставлено в даний час – інтерпретація w і b як ваги та упередження, σ функція, яка ховається у фоновому режимі, вибір архітектури мережі, *MNIST* тощо.

Виявляється, можемо зрозуміти надзвичайно багато, ігноруючи більшу частину цієї структури і просто зосередившись на аспекті мінімізації. Тож наразі забудемо все про конкретну форму функції витрат, зв'язок з нейронними мережами тощо. Натомість збираємось уявити, що просто дали функцію безлічі змінних, і хочемо мінімізувати цю функцію. Розробимо техніку, яка називається градієнтним спуском, яка може бути використана для вирішення таких задач мінімізації. Тоді повернемося до конкретної функції, яку хочемо мінімізувати для нейронних мереж.

При спробі мінімізувати якусь функцію $C(v)$ (це може бути будь-яка дійсна функція багатьох змінних $v = v_1, v_2,\ldots$) звертаємо увагу, що було замінено в w і b позначення за v, щоб підкреслити, що це може бути будь-яка функція – вже конкретно не думаємо в контексті нейронних мереж. Необхідно звести до мінімуму C (v) це допомагає уявити C (рис. 2.6), як функцію лише двох змінних, які будемо називати v_1 і v_2.

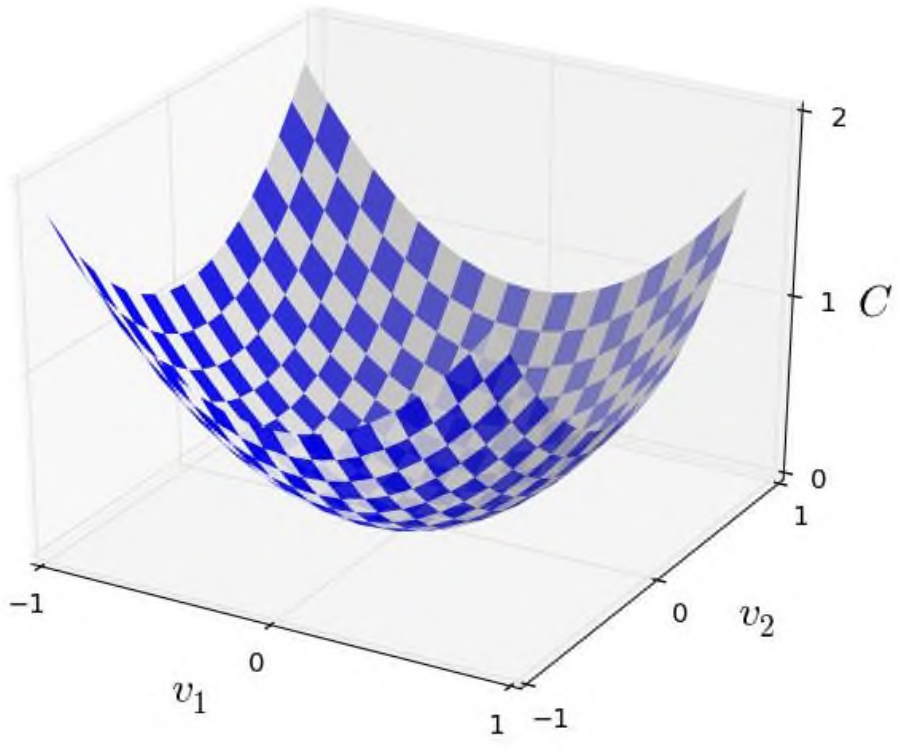

Рис. 2.6. Пошук мінімуму функції методом градієнтного спуску

Треба знайти, де C досягає свого глобального мінімуму. Тепер, звичайно, для накладеної вище функції можемо окулювати графік і знайти мінімум. У

цьому сенсі, можливо, зображено трохи надто просту функцію. Загальна функція C може бути складною функцією багатьох змінних, і, як правило, потім неможливо на графіку знайти мінімум.

Одним із способів атакувати проблему є використання числення, щоб спробувати аналітично знайти мінімум. Можна було б обчислити похідні, а потім спробувати використовувати їх, щоб знайти місця, де C є екстремумом. З деякою удачею, яка може спрацювати, коли C є функцією лише однієї або декількох змінних. Але це перетвориться на жах, коли у нас буде набагато більше змінних. А для нейронних мереж часто хочемо набагато більше змінних – найбільші нейронні мережі мають функції витрат, які надзвичайно складно залежать від мільярдів ваг та упереджень. Використання даного числення для мінімізації просто не буде працювати.

Після твердження, що отримаємо розуміння, уявляючи C як функція всього двох змінних. Представимо C як функцію двох змінних. Просто трапляється, що іноді ця картина руйнується. Грамотне розмірковування про математику часто передбачає жонглювання кількома інтуїтивними картинками, вивчення того, коли доречно використовувати кожне зображення, а коли ні.

Саме тому числення не працює. Є чудова аналогія, яка пропонує алгоритм, який працює досить добре. Потрібно почати з того, що думаємо про свою функцію як про долину. Уявляємо, як куля котиться по схилу долини. Наш повсякденний досвід говорить нам, що м’яч врешті-решт скотиться на дно долини. Можливо, можемо використати цю ідею як спосіб знайти мінімум для функції. У разі, якщо випадково вибирали початкову точку для (уявної) кулі, а потім імітували рух кульки, коли вона котилася вниз на дно долини. Можна було бзробити це моделювання просто обчислюючи похідні (і, можливо, деякі інші похідні) C – ці похідні сказали б усе, що потрібно знати про місцеву "форму" долини, а отже, як має рухатися наша куля.

Виходячи з представленого доказу можна припустити, яким чином слід намагатись записати рівняння руху Ньютона для кулі, враховуючи вплив тертя та сили тяжіння тощо. Насправді, не будемо сприймати аналогію з рухомою кулькою настільки серйозно – розробляємо алгоритм для мінімізації C , не

розробляючи точного моделювання законів фізики. Вид м'яча з погляду покликаний стимулювати нашу уяву, а не стримувати наше мислення. Тож замість того, щоб вникати у всі безладні подробиці фізики, давайте просто запитаємо себе: якби ми змогли б скласти свої власні закони фізики, диктуючи кульці, як воно має рухатися, який закон чи закони чи можемо обрати такий рух, щоб м'яч завжди котився на дно долини.

Щоб зробити це питання більш точним, давайте подумаємо, що відбувається, коли рухаємо м'яч невеликою кількістю Δv_1 в v_1 напрямку, і невелика сума Δv_2 в v_2 напрямку. Обчислення говорить це C змінюється наступним чином:

$$\Delta C \approx \frac{\partial C \Delta v_1}{\partial v_1} + \frac{\partial C \Delta v_2}{\partial v_2}..$$

Знайдемо спосіб вибору Δv_1 і Δv_2, щоб зробити ΔC негативним, тобто виберемо їх, щоб куля котилася вниз у долину, щоб зрозуміти, як зробити такий вибір. Це допомагає визначитися $\Delta v \Delta v$ бути вектором змін в v:

$$\Delta v \equiv \left(\Delta v_1, \Delta v_2\right)^T,$$

де T – це знову операція транспонування, перетворюючи вектори рядків у вектори стовпців.

Також визначимо градієнт C бути вектором часткових похідних, $\left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$. Позначимо вектор градієнта через$\nabla C.\nabla C.$, тобто:

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T.$$

Перепишемо зміни ΔC з точки зору Δv і градієнт, ∇C. Однак перед тим, як дійти до цього, хочемо пояснити, що іноді змушує людей зависати на градієнті.

∇C як єдиний математичний об'єкт – вектор, визначений вище, який, як правило, записується за допомогою двох символів. З цієї точки зору, ∇ це лише шматочок нотаційного розмахування прапором, який каже "∇C є градієнтним вектором ", ∇ може розглядатися як самостійна математична сутність (наприклад,

як диференціальний оператор). З цими визначеннями вираз для ΔC можна переписати як:

$$\Delta C \approx \nabla C \cdot \Delta v.$$

Це рівняння допомагає пояснити, чому ∇C. називається вектором градієнта: ∇C, стосується змін у v до змін у C, так само, як очікували зробити щось, що називається градієнтом. Але що справді захоплює рівняння, це те, що воно дозволяє побачити, як вибрати Δv щоб зробити ΔC негативним. Зокрема, припустимо, що обираємо:

$$\Delta v = -\eta \nabla C,$$

де η – це невеликий, позитивний параметр (відомий як швидкість навчання).

Тоді рівняння говорить, що:

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2.$$

Оскільки $\|\nabla C\|^2 \geq 0$ гарантує $\Delta C \leq 0$, тобто C завжди зменшиться, ніколи не збільшиться, якщо змінимось v за формулою.

Потім знову використаємо це правило оновлення, щоб зробити ще один крок. Якщо продовжуватимемо робити це знову і знову, будемо зменшувати C поки не досягнемо глобального мінімуму. Підводячи підсумок, способу роботи алгоритму градієнтного спуску полягає в багаторазовому обчисленні градієнта ∇C, а потім виборі рухатися у зворотному напрямку, «падаючи» по схилу долини. Можемо візуалізувати це на рисунку 2.7.

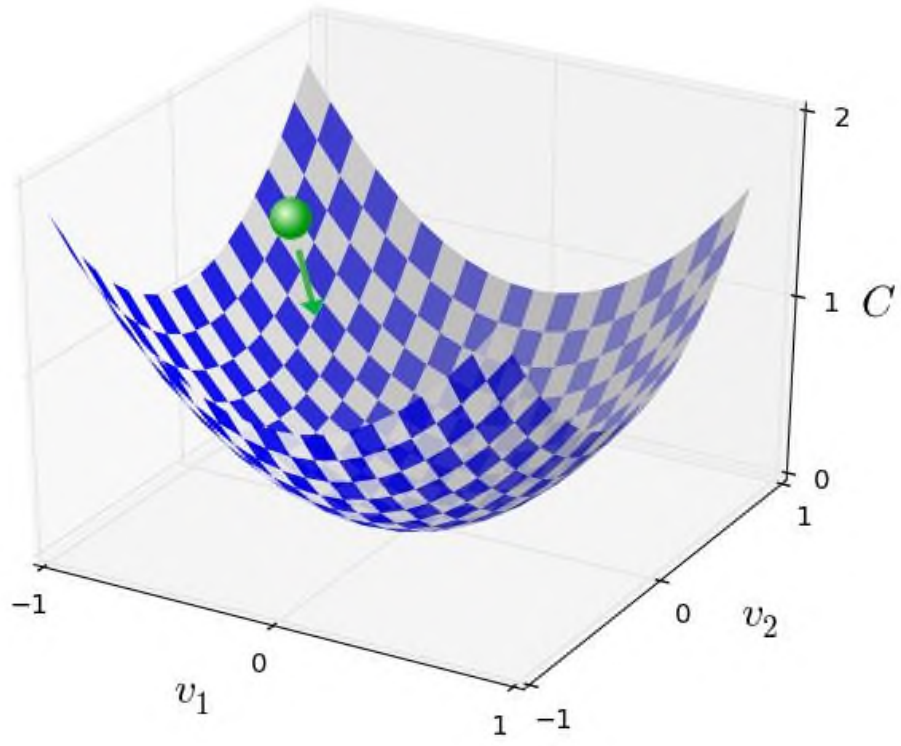

Рис. 2.7. Візуалізація методу градієнтного спуску

За цим правилом градієнтний спуск не відтворює реального фізичного руху. У реальному житті куля має імпульс, і цей імпульс може дозволити їй котитися по схилу або навіть (на мить) котитися вгору. Лише після встановлення ефекту тертя м’яч гарантовано котиться вниз у долину. Навпаки, наше правило вибору Δv просто каже "спустись, прямо зараз". Це все ще досить гарне правило для пошуку мінімуму.

Щоб градієнтний спуск працював правильно, потрібно вибрати швидкість навчання η бути достатньо малим, щоб рівняння для ΔC було хорошим наближенням. Якщо цього не зробимо, можемо закінчити $\Delta C > 0$. У той же час не хочемо, щоб η було замалим, оскільки це внесе зміни Δ, і, отже, алгоритм градієнтного спуску буде працювати дуже повільно. У практичних реалізаціях, η часто змінюється так, що рівняння для ΔC залишається хорошим наближенням, але алгоритм не надто повільний. Як це працює, побачимо пізніше.

У наведеному прикладі C є функцією лише двох змінних. Але насправді все працює так само добре навіть тоді, коли C є функцією багатьох інших

змінних. Припустимо, зокрема, що C є функцією мм змінні, $v_1,\ldots, v_m$. Потім зміна ΔC в C виробляється невеликою зміною $\Delta v = (\Delta v_1,\ldots, \Delta v_m)^T$:

$$\Delta C \approx \nabla C \cdot \Delta v,$$

де градієнт ∇C є вектором:

$$\nabla C \equiv (\partial C/\partial v_1,\ldots, \partial C/\partial v_m)^{\mathrm{T}}.$$

Як і для двох змінних випадків, можемо вибрати $\Delta v = -\eta \nabla C$ і гарантувати, що наш (приблизний) вираз для ΔC буде негативним. Це дає спосіб слідувати градієнту до мінімуму, навіть коли C є функцією багатьох змінних шляхом неодноразового застосування правила оновлення:

$$v \rightarrow v' = v - \eta \nabla C.$$

Це дає спосіб багаторазової зміни позиції v для того, щоб знайти мінімум функції C. Правило не завжди працює – кілька речей можуть піти не так і перешкодити градієнтному спуску знайти загальний мінімум C. Але на практиці градієнтний спуск часто працює надзвичайно добре, і в нейронних мережах виявимо, що це потужний спосіб мінімізувати функцію витрат, а значить, допомогти мережі навчитися.

Дійсно, навіть існує сенс, коли градієнтний спуск є оптимальною стратегією для пошуку мінімуму. Припустимо, що намагаємось зробити крок Δv в положенні так, щоб зменшуватися C так багато, як тільки можливо. Це еквівалентно мінімізації $\Delta C \approx \nabla C \cdot \Delta v$. Обмежимо розмір переїзду таким чином $\|\Delta v\| = \epsilon$ для деяких невеликих фіксованих $\epsilon > 0$. Іншими словами, потрібен рух, який є невеликим кроком фіксованого розміру намагається знайти напрямок руху, який зменшується C так багато, як тільки можливо. Можна довести, що вибір Δv, який мінімізує $\nabla C \cdot \Delta v$, де $\eta = \epsilon / \|\nabla C\|$ визначається обмеженням розміру $\|\Delta v\| = \epsilon$. Отже,

градієнтний спуск можна розглядати як спосіб зробити невеликі кроки у напрямку, який робить найбільше для негайного зменшення C.

2.3. Модифікація методу варіації градієнтного спуску

В наукових роботах досліджували багато варіацій градієнтного спуску, включаючи варіанти, які більш точно імітують справжній фізичний м'яч. Ці варіації, що імітують кульки, мають деякі переваги, але також мають головний недолік: виявляється, необхідно обчислити другі часткові похідні C, а це може коштувати досить дорого. Щоб зрозуміти, чому це дорого, припустимо, хочемо обчислити всі другі часткові похідні $\partial^2 C / \partial v_j \partial v_k$. Якщо таких мільйон v_j, тоді потрібно буде обчислити щось на зразок трильйону (тобто мільйона в квадраті) других часткових похідних.

Насправді, більше як пів трильйона і це буде обчислювально дорого. З огляду на це, є трюки для уникнення такого роду проблем, і пошук альтернатив градієнтному спуску є активною сферою дослідження. Але використовуватимемо градієнтний спуск (і його варіації) як основний підхід до навчання в нейронних мережах.

Для навчання в нейронній мережі можемо застосувати градієнтний спуск. Ідея полягає у використанні градієнтного спуску для пошуку ваг w_k та упередженості b_l які мінімізують витрати в рівнянні. Щоб побачити, як це працює, давайте повторимо правило оновлення градієнтного спуску, ваги та упередження замінять змінні v_j. Іншими словами, наша "позиція" тепер має складові w_k і b_l і вектор градієнта ∇C має відповідні компоненти $\partial C/\partial w_k$ і $\partial C/\partial b_l$. Випишемо правило оновлення градієнтного спуску з точки зору компонентів.

Повторно застосовуючи це правило оновлення, можемо "скотитися з гори" і, сподіваємось, знайти мінімум функції витрат. Іншими словами, це правило, яке можна використовувати для навчання в нейронній мережі.

Існує ряд проблем у застосуванні правила градієнтного спуску. Глибше розглянемо їх у наступних розділах. Але поки що є одна проблема. Щоб зрозуміти, в чому проблема, потрібно подивитися назад на квадратну вартість у

рівнянні. Звернемо увагу, що ця функція витрат має вигляд $C = ln\sum xC_x$ тобто це середнє значення витрат $C_x \equiv \|y(x) - a\|^2$ для індивідуальних навчальних прикладів. На практиці для обчислення градієнта ∇C потрібно обчислити градієнти ∇Cx окремо для кожного навчального матеріалу x, а потім усереднити їх, $\nabla C = C = ln\sum x\nabla C_x$. На жаль, коли кількість навчальних матеріалів дуже велика, це може зайняти багато часу, і навчання, таким чином, відбувається повільно.

Ідею, яка називається стохастичним градієнтним спуском, можна використовувати для пришвидшення навчання. Ідея полягає в оцінці градієнта ∇C за допомогою обчислень ∇C_x для невеликої вибірки випадково вибраних навчальних матеріалів. Посереднюючи для цієї невеликої вибірки, виявляється, що можливо швидко отримати хорошу оцінку справжнього градієнта ∇C, і це допомагає прискорити градієнтний спуск, а отже і навчання.

Щоб зробити ці ідеї більш точними, стохастичний градієнтний спуск працює шляхом випадкового виділення невеликої кількості випадково вибраних навчальних матеріалів. Позначимо ці випадкові вхідні дані для навчання $X_1, X_2, \ldots, X_m$ і передаємо їх як міні-пакет. Розміо вибірки рукописного тексту є досить великим, очікуємо, що середнє значення ∇CX_j буде приблизно дорівнювати середньому по всіх ∇Cx, підтверджуючи, що можемо оцінити загальний градієнт, обчислюючи градієнти лише для випадково обраної міні-партії.

Припустимо, щоб явно пов’язати це з навчанням у нейронних мережах w_k і b_l позначаємо ваги та упередження в даній нейронній мережі. Потім стохастичний градієнтний спуск працює, вибираючи випадково обрану міні-партію навчальних матеріалів, і тренуючись із ними. Потім вибираємо іншу довільно обрану міні-партію і тренуємось із ними. І так далі, поки не вичерпається навчальний матеріал, який завершує епоху навчання. На цьому етапі починаємо спочатку з нової епохи навчання.

До речі, варто зазначити, що домовленості різняться щодо масштабування функції витрат та міні-пакетного оновлення ваг та упереджень. У рівнянні масштабували загальну функцію витрат на коефіцієнт n. Інколи опускають n,

підсумовуючи витрати на окремі приклади навчання замість усереднення. Це особливо корисно, коли загальна кількість прикладів тренувань не відома заздалегідь. Це може статися, якщо, наприклад, у реальному часі генерується більше навчальних даних. І, подібним чином, правила оновлення міні-пакетного оновлення іноді опускають $1/m$ закінчувати фронт сум. Концептуально це мало що робить, оскільки це еквівалентно масштабуванню рівня навчання η. Але при детальному порівнянні різних робіт варто стежити.

Можемо вважати стохастичний градієнтний спуск подібним до політичного опитування: набагато простіше взяти вибірку для невеликої міні-партії, ніж застосувати градієнтний спуск до повної партії, так само, як проведення опитування легше, ніж проведення повних виборів. Наприклад, якщо у нас є навчальний набір розміру $n = 60000$, як у *MNIST*, і вибрно розмір міні-партії (скажімо) $m = 10$, це означає, що отримаємо коефіцієнт 6 000 прискорення в оцінці градієнта. Звичайно, оцінка не буде ідеальною – будуть статистичні коливання – але вона не повинна бути ідеальною: все, що насправді турбує, – рухатися в загальному напрямку, який допоможе зменшити C , а це означає, що не потрібно точне обчислення градієнта. На практиці стохастичний градієнтний спуск є загальновживаною і потужною технікою навчання в нейронних мережах, і це основа для більшості методів навчання.

У нейронних мережах вартість C це, звичайно, функція багатьох змінних – усіх ваг та упереджень – і тому в якомусь сенсі визначає поверхню у дуже високомірному просторі. Доволі важко інтерпретувати та візуалізувати всі ці додаткові виміри, багато людей не можуть мислити в чотирьох вимірах, не кажучи вже про п'ять (або п'ять мільйонів). Навіть більшість професійних математиків не можуть візуалізувати чотири виміри особливо добре. Натомість фокус, який вони використовують, полягає у розробці інших способів відображення того, що відбувається. Це саме те, що зробили вище: використовували алгебраїчне (а не візуальне) подання ΔC, щоб зрозуміти, як рухатися так, щоб зменшуватися C . Люди, які добре вміють мислити у великих вимірах, мають ментальну бібліотеку, що містить багато різних методів у цьому напрямку, цей алгебраїчний трюк – лише один із прикладів. Ці методи можуть

мати не ту простоту, до якої звикли при візуалізації три виміри, але як тільки створили бібліотеку таких методів, можна досить добре мислити у великих вимірах.

2.4. Впровадження мережі для класифікації цифр

При написанні програми, щоб розпізнавати рукописні цифри, використовуємо стохастичний градієнтний спуск та навчальні дані *MNIST*.

Дані *MNIST* розділені на 60 000 навчальних зображень та 10 000 тестових зображень. Це офіційний опис *MNIST*. Для роботи програми треба розділити дані трохи інакше. Залишимо тестові зображення як є, але розділимо навчальний набір *MNIST* із 60 000 зображень на дві частини: набір з 50 000 зображень, який будемо використовувати для навчання нейронної мережі, та окремий набір для перевірки 10000 зображень. Не будемо використовувати дані перевірки, але пізніше знайдемо це корисним для з'ясування того, як встановити певні гіперпараметри нейронної мережі – такі речі, як швидкість навчання тощо, які не є безпосередньо не обраними нашим алгоритмом навчання. Хоча дані перевірки не є частиною вихідної специфікації *MNIST*, зазвичай використовують *MNIST* таким чином, а використання даних перевірки є загальним у нейронних мережах. Відтепер під фразою "навчальні дані *MNIST*" потрібно мати на увазі набір даних 50 000 зображень, а не вихідний набір даних 600 000 зображень.

2.4.1. Представлення згорткової мережі

Розпізнавання цифр робиться за допомогою мереж, в яких сусідні мережеві шари повністю зв’язані один з одним. Тобто кожен нейрон у мережі пов’язаний з кожним нейроном у сусідніх шарах (рис. 2.7).

Рис. 2.8. Нейронна мережа з прихованими шарами

Зокрема, для кожного пікселя на вхідному зображенні закодували інтенсивність пікселя як значення для відповідного нейрона у вхідному шарі. Для 28 × 28 піксельні зображення, які використовували, це означає, що їх має наша мережа 784 = 28 × 28 вхідні нейрони. Потім навчили ваги та упередженості мережі, щоб результат роботи мережі з високою вірогідністю правильно визначав вхідне зображення: '0', '1', '2', ..., '8' або '9'.

Попередні мережі працювали достатньо добре: працювали та отримали точність класифікації, що перевищує 98 відсотків, використовуючи навчальні та тестові дані з набору рукописних цифр *MNIST*. Але після аналізу використання мереж із повністю зв’язаними шарами для класифікації зображень є не доречним. Причина в тому, що така мережева архітектура не враховує просторову структуру зображень. Наприклад, розглядаються вхідні пікселі, які знаходяться далеко один від одного і зближуються на однаковому рівні. Натомість про такі поняття просторової структури слід робити висновки з навчальних даних. Але, якщо замість того, щоб розпочати з мережевої архітектури, яка є *tabula rasa*, використали архітектуру, яка намагається скористатися перевагами просторової структури, то можемо перейти до згорткових нейронних мереж. Витоки згорткових нейронних мереж сягають 1970-х років. Але основоположним документом, що встановлює сучасну тему згорткових мереж, був документ 1998 року, "Навчання на основі градієнта застосовується до розпізнавання

документів", Янн ЛеКун, Леон Ботту, Йошуа Бенджо та Патрік Хаффнер. З того часу ЛеКун зробив цікаве зауваження щодо термінології згорткових мереж: "Нейронне натхнення в таких моделях, як згорткові мережі, є дуже слабким. Ось чому використовується термін згорткові мережі, а не згорткові нейронні мережі, і чому називаються вузли одиницями, а не нейронами". Незважаючи на це зауваження, згорткові мережі використовують багато тих самих ідей, що і нейронні мережі, які вивчали дотепер: такі ідеї, як зворотне поширення, градієнтний спуск, регуляризація, нелінійні функції активації тощо. І тому потрібно слідувати загальноприйнятій практиці і вважати їх різновидом нейронних мереж. Потрібно використовувати терміни "згорткова нейронна мережа" та "згорткова мережа (робота)" як взаємозамінні. Також треба використовувати терміни "[штучний] нейрон" та "одиниця" як взаємозамінні. Ці мережі використовують спеціальну архітектуру, яка особливо добре адаптована для класифікації зображень. Використання цієї архітектури дозволяє швидко розвивати конволюційні мережі. Це, у свою чергу, допомагає тренувати глибокі багатошарові мережі, які дуже добре класифікують зображення. Сьогодні в більшості нейронних мереж для розпізнавання зображень використовуються глибокі згорткові мережі або якийсь близький варіант.

Світові нейронні мережі використовують три основні ідеї: місцеві рецептивні поля, спільні ваги та об’єднання. Протрібно проаналізувати кожну з цих ідей.

Місцеві рецептивні поля: у повністю з'єднаних шарах, показаних раніше, входи зображувались як вертикальна лінія нейронів. У згортковій мережі це допомагає мати замість входів як 28 × 28 квадрат нейронів, значення яких відповідають 28 × 28 інтенсивність пікселів, яку використовують як вхідні дані (рис. 2.9).

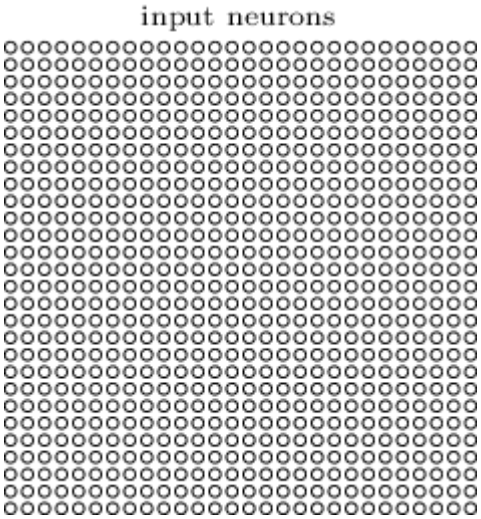

Рис. 2.9. Матрия вхідних нейронів

Необхідно підключити вхідні пікселі до шару прихованих нейронів. Але не треба підключати кожен вхідний піксель до кожного прихованого нейрона. Натомість потрібно встановити зв’язки лише в невеликих локалізованих регіонах вхідного зображення.

Точніше кажучи, кожен нейрон у першому прихованому шарі буде з'єднаний з невеликою областю вхідних нейронів, як, наприклад, 5 × 5 регіону, що відповідає 25 вхідним пікселям. Отже, для конкретного прихованого нейрона можемо мати зв’язки, які виглядають, як на рисунку 2.10.

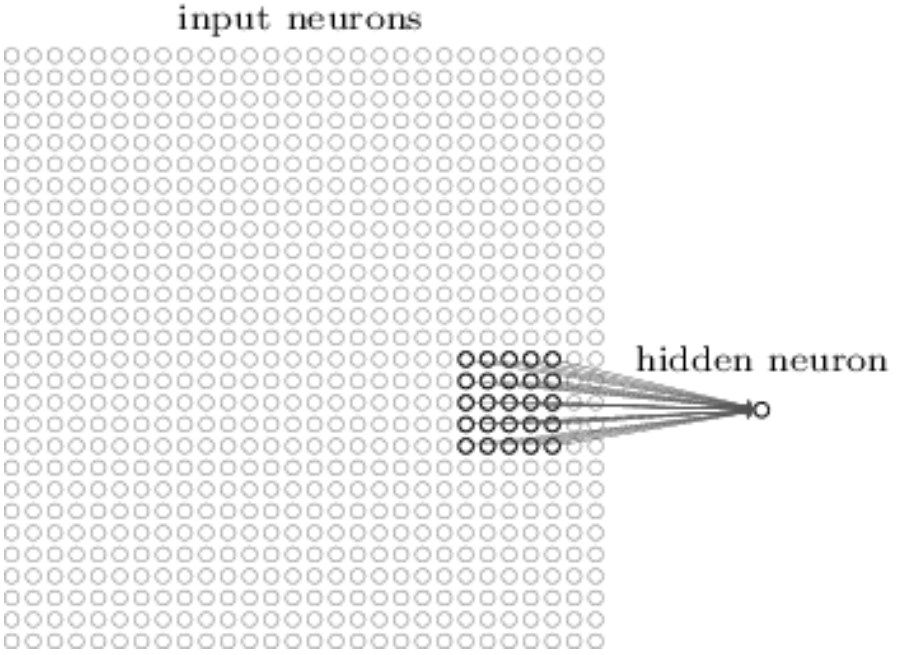

Рис. 2.10. Приховані нейрони

Ця область на вхідному зображенні називається місцевим рецептивним полем для прихованого нейрона. Це маленьке віконце на вхідних пікселях. Кожне з'єднання вивчає вагу. І прихований нейрон також вивчає загальну упередженість. Можна думати про цей конкретний прихований нейрон як про навчання аналізу його конкретного місцевого рецептивного поля.

Потім треба пересунути місцеве сприйнятливе поле по всьому вхідному зображенню. Для кожного локального рецептивного поля існує перший прихований нейрон у першому прихованому шарі. Щоб наочно проілюструвати це, необхідно почати з місцевого рецептивного поля у верхньому лівому куті (рис. 2.11).

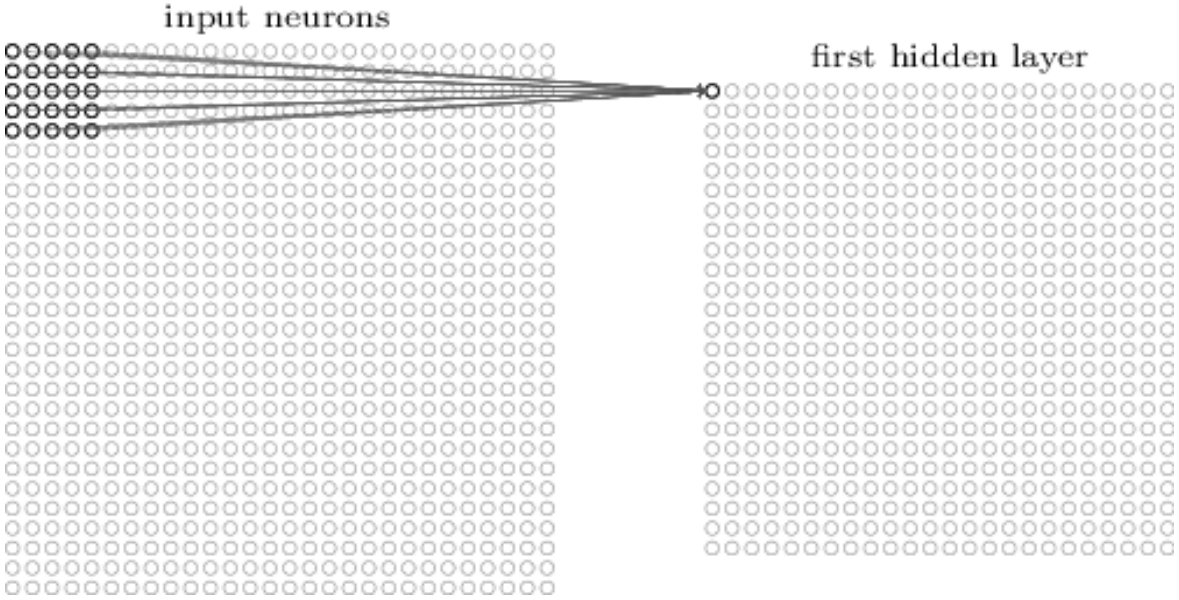

Рис. 2.11. Візуалізація прихованих нейронів

Потім треба зсунути місцеве рецептивне поле на один піксель праворуч (тобто на один нейрон), щоб з’єднатися з другим прихованим нейроном (рис. 2.12).

І так далі, нарощуючи перший прихований шар. Потрібно маті на увазі, що якщо є 28 × 28 вхідне зображення, і 5 × 5 місцеві рецептивні поля, тоді вони будуть 24 × 24 нейрони в прихованому шарі. Це пов’язано з тим, що можливо перемістити лише місцеве сприйнятливе поле на 23 нейрони поперек (або 23 нейрони вниз), перед зіткненням з правою стороною (або внизу) вхідного зображення.

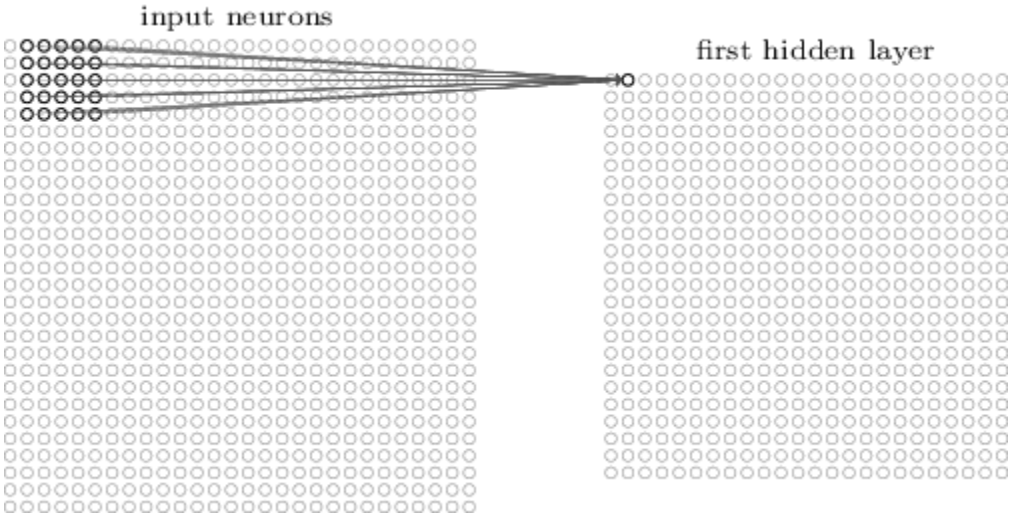

Рис. 2.12. Візуалізація зсуву по матриці прихованих нейронів

Зображено, що місцеве сприйнятливе поле переміщується на один піксель за раз. Інколи використовується інша довжина кроку. Наприклад, можливо перемістити місцеве рецептивне поле на 2 пікселі праворуч (або вниз), в цьому випадку б сказали використовувати довжину кроку 2. Потрібно здебільшого дотримуватимемося довжини кроку 1, але варто знати, що іноді можливо експериментувати з різною довжиною кроку.

2.4.2. Спільні ваги та упередження:

Прихований нейрон має упередження і 5 × 5 ваги, пов'язані з місцевим рецептивним полем. Потрібно використовувати однакові ваги та упередження для кожного з них 24 × 24 приховані нейрони. Іншими словами, для *j k*-го прихованого нейрона, результат:

$$\sigma\left(b+\ \sum_l \sum_j w_{l,m} a_{j+l,k+m}\right),$$

де σ – функція нервової активації,

b – загальне значення для упередженості,

$w_{l,m}$ – 5 × 5 масив спільних ваг,

$a_{x,\,y}$ – позначення активації входу в положенні x, y.

Це означає, що всі нейрони першого прихованого шару виявляють абсолютно однакову ознаку. Зазвичай необхідно застосовувати один і той же детектор функцій на зображенні. Маючи на увазі, що згорткові мережі добре пристосовані до незмінності перекладу зображень: трохи перемістити зображення кота, і це все ще образ кота. Фактично, для проблеми класифікації цифр *MNIST,* які вивчали, зображення відцентровані та нормалізовані за розміром. Отже, *MNIST* має меншу незмінність перекладу, ніж зображення, знайдені "в дикій природі". Тим не менш, такі функції, як краї та кути, можуть бути корисними для більшої частини вхідного простору.

З цієї причини іноді називають карту від вхідного шару до прихованого шару картою об’єктів. Вагами, що визначають карту об’єктів, називають спільними вагами. Та називають упередження, визначаючи карту об’єктів таким чином, спільним упередженням. Деколи мають на увазі, що спільні ваги та зміщення визначають ядро або фільтр.

Структура мережі, яку було описано, може виявити лише один вид локалізованої функції. Для розпізнавання зображень знадобиться більше ніж одна карта об’єктів. І отже, повний згортковий шар складається з декількох різних функціональних карт (рис. 2.13).

У наведеному прикладі є 3 особливості карт. Кожна карта об'єктів визначається набором 5 × 5 спільних важелів та єдиного спільного упередження. В результаті мережа може виявити 3 різні типи функцій, причому кожна функція виявляється на всьому зображенні.

Відображено 3 карти функцій, щоб зробити схему простою. Однак на практиці згорткові мережі можуть використовувати більше (і, можливо, набагато більше) функціональних карт.

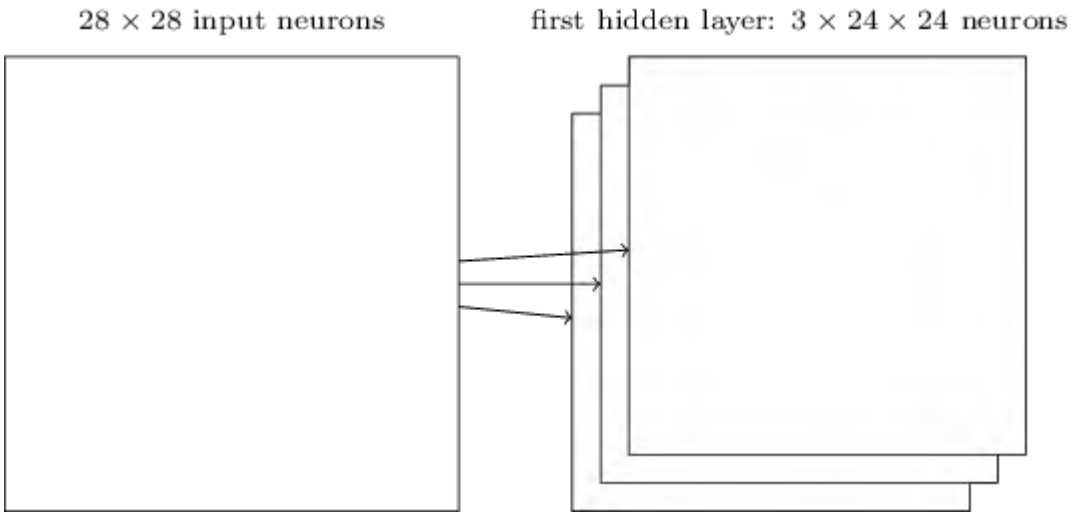

Рис. 2.13. Група функціональних карт

Використовується одна з ранніх згорткових мереж, *LeNet*-566 карти об'єктів, кожна з яких пов'язана з 5 × 5 місцевим рецептивним полем, для розпізнавання цифр *MNIST*. Отже, приклад, проілюстрований вище, насправді досить близький до *LeNet*-5. У прикладах, які розробимо далі, будемо використовувати згорткові шари з 20 і 40 карт особливостей. Ілюстровані карти функцій походять від остаточної згорткової мережі, яку тренуємо (рис. 2.14).

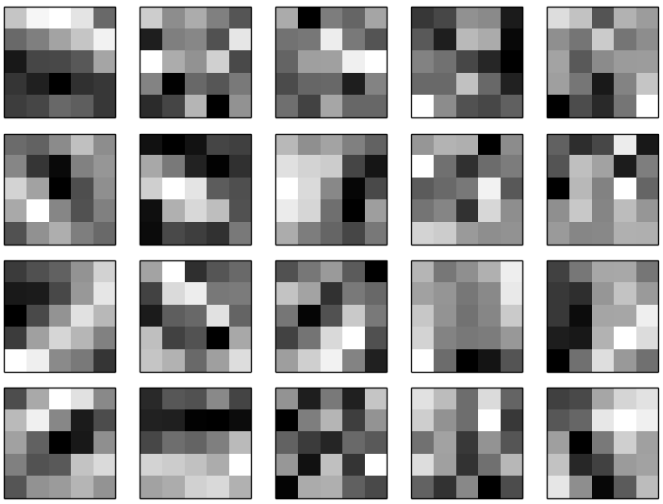

Рис. 2.14. Карта з 20 зображень для навчання

20 зображень відповідають 20 картам функцій (або фільтрам, або ядрам). Кожна карта представлена як 5 × 5 блоків зображення, що відповідає 5 × 5 важелів в місцевому рецептивному полі. Білі блоки означають меншу (як правило, більш негативну) вагу, тому карта об’єктів менше реагує на відповідні

вхідні пікселі. Більш темні блоки означають більшу вагу, тому карта об’єктів реагує більше на відповідні вхідні пікселі. Зображення вище показують тип особливостей, на які реагує згортковий шар.

Тут існує просторова структура, яка перевищує очікувані випадки: багато функцій мають чіткі підрегіони світла та темряви. Це показує, що дана мережа насправді вивчає речі, пов’язані з просторовою структурою. Однак, крім цього, важко зрозуміти, чому навчаються ці детектори функцій. Зараз, не використовуються Фільтри Габораякі, які раніше використовувались у багатьох традиційних підходах до розпізнавання зображень. На даний момент багато роботи проводитсься над кращим розумінням особливостей, засвоєних конволюційними мережами.

Великою перевагою спільного використання ваг та упереджень є те, що це значно зменшує кількість параметрів, що беруть участь у згортковій мережі. Для кожної карти об’єктів потрібно 25 = 5 × 5 спільних важелів, плюс одне спільне упередження. Тож кожна карта об’єктів вимагає 26 параметрів. Якщо мати 20 об'єктів карти, що загалом 20 × 26 = 520 параметрів, що визначають згортковий шар. Для порівняння, якщо б був повністю зв’язаний перший шар, з 784 = 28 × 28 вхідних нейронів, і відносно скромний 30 прихованих нейронів, які використовували у багатьох прикладах, наведених раніше. Це всього 784 × 30 важелів, плюс додаткові 30 упередженостей, загалом 23 550 параметри. Іншими словами, повністю пов’язаний шар мав би більше ніж в 40 разів більше параметрів, ніж згортковий шар.

Не потрібно здійснювати прямого порівняння між кількістю параметрів, оскільки ці дві моделі різні за суттю. Але інтуїтивно можна припустити ймовірно, що використання інваріантності перекладу згортковим шаром зменшить кількість параметрів, необхідних для отримання такої ж продуктивності, як і повністю підключена модель. Це, в свою чергу, призведе до швидшого навчання для згорткової моделі і, зрештою, допоможе побудувати глибокі мережі з використанням згорткових шарів.

2.4.3. Об'єднання шарів

Згорткових шари, які були описані вище також містять шари об'єднання. Об'єднання шарів зазвичай використовується відразу після згорткових шарів. Шари об’єднання роблять спрощення інформації у вихідних даних із згорткового шару.

Зокрема, використовують "карту об'єктів", щоб означати не функцію, обчислену згортковим шаром, а активацію прихованих нейронів, що виводяться із шару. Вихід із згорткового шару і готує стиснуту карту об'єктів. Наприклад, кожен блок у шарі об'єднання може узагальнювати область, скажімо, 2 × 2 нейрони попереднього шару. Як конкретний приклад, однією із загальноприйнятих процедур об'єднання є макс-пул. При максимальному об'єднанні блок об'єднання просто видає максимальну активацію в 2 × 2 область введення, як показано на діаграмі рис. 2.15.

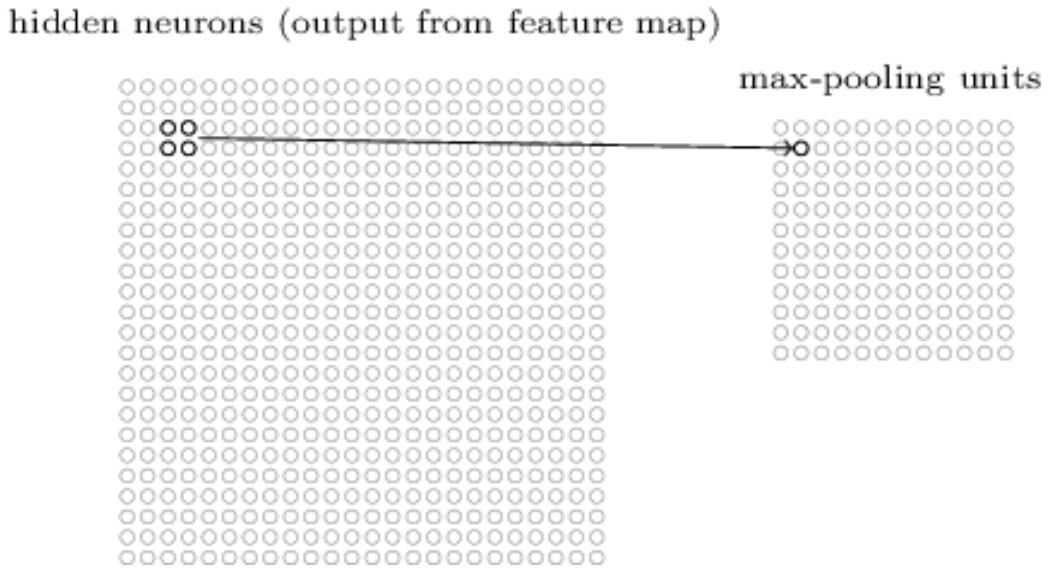

Рис. 2.15. Об’єднання шарів 2 × 2

Необхідно звернути увагу на те, що оскільки маємо 24 × 24 нейрони, які виводяться із згорткового шару, після об'єднання маємо 12 × 12 нейронів.

Як згадувалося вище, згортковий шар зазвичай включає більше, ніж одну карту об'єктів. Потрібно застосовати макс-пул до кожної карти об’єктів окремо. Отже, якби було три карти функцій, комбіновані згорткові та максимально об’єднані шари виглядали б так, як на рис. 2.16.

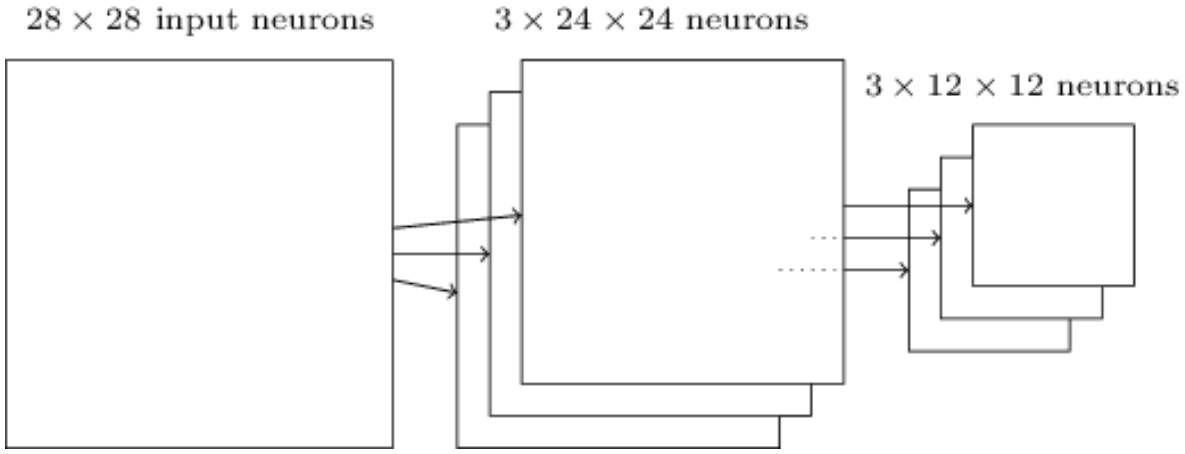

Рис. 2.16. Об’єднання шарів нейронів в 3 картах

Можна розглядати макс-пулінг як спосіб для мережі запитати, чи є дана функція в будь-якому місці в області зображення. Потім він викидає точну позиційну інформацію. Інтуїція полягає в тому, що після того, як об’єкт знайдено, його точне розташування не так важливо, як його приблизне розташування щодо інших об’єктів. Великою перевагою є те, що набагато менше об’єднаних функцій, і це допомагає зменшити кількість параметрів, необхідних для наступних шарів.

Макс-пулінг – не єдина техніка, яка використовується для об’єднання. Інший поширений підхід відомий як об'єднання *L2*. Тут замість того, щоб взяти максимальну активацію 2 × 2 в області нейронів беруть квадратний корінь із суми квадратів активацій в 2 × 2 регіону. Хоч деталі різні, інтуїція схожа на макс-пул: об'єднання *L2* – це спосіб конденсації інформації із згорткового шару. На практиці обидві методики широко використовуються. Також іноді використовуються інші типи операцій об'єднання. Якщо намагатися оптимізувати продуктивність, можна використовувати дані перевірки, щоб порівняти кілька різних підходів до об’єднання та вибрати підхід, який найкраще працює.

Тепер необхідно об’єднати всі ці ідеї, щоб сформувати цілісну згорткову нейронну мережу. Це схоже на архітектуру, яка була розглянута, але має доданий шар з 10 вихідних нейронів, відповідні 10 можливих значень для цифр *MNIST* ('0', '1', '2' тощо).

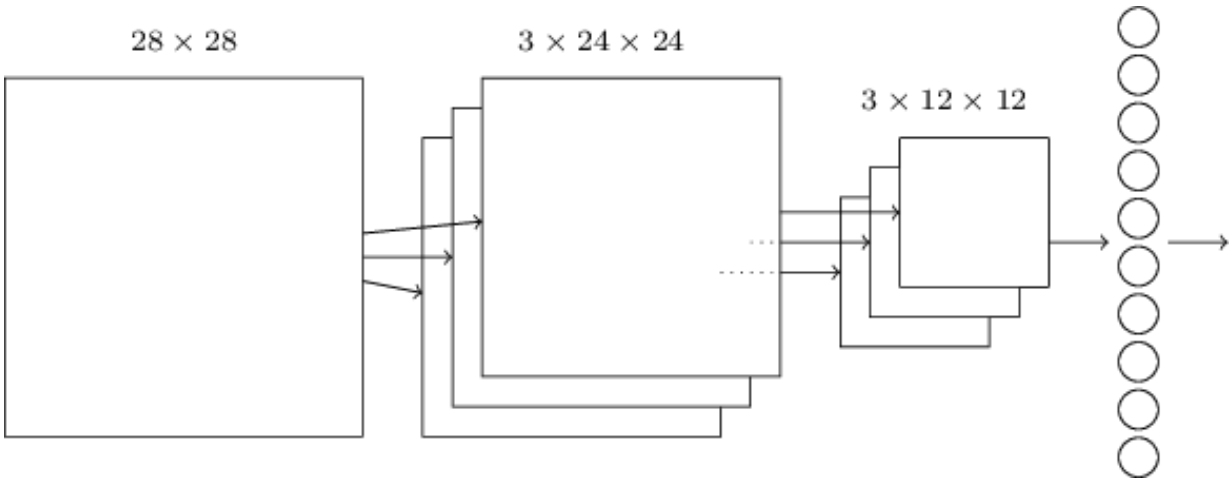

Рис. 2.17. Згортання та об’єднання нейронів

Мережа починається з 28 × 28 вхідних нейронів, які використовуються для кодування інтенсивності пікселів для зображення *MNIST*. Потім слідує згортковий шар з використанням 5 × 5 місцевого рецептивного поля та 33 особливості карт. Результат – шар 3 × 24 × 24 прихованих особливостей нейронів. Наступним кроком є шар об'єднання, застосований до 2 × 2 у кожному з регіонів та 3 особливості карт. Результат – шар 3 × 12 × 12 – прихована особливість нейронів.

Кінцевий рівень з'єднань у мережі – це повністю пов'язаний рівень. Тобто цей шар з’єднує кожен нейрон з максимально об’єднаного шару з кожним із 10 вихідних нейронів. Це повністю пов'язана архітектура, однак необхідно зауважити, що на діаграмі вище використовувалась одна стрілка для простоти, а не для показу всіх з’єднань.

Ця згорткова архітектура досить сильно відрізняється від інших вище описаних архітектур. Але загальна картина схожа: мережа, що складається з безлічі простих одиниць, поведінка яких визначається вагою та упередженістю. І загальна мета залишається незмінною: використовувати навчальні дані для тренування ваг та упереджень мережі, щоб мережа добре виконувала класифікацію вхідних цифр.

Мережу необхідно навчати використовуючи стохастичний градієнтний спуск та зворотне розповсюдження. Здебільшого це відбувається точно так само,

як і в попередніх навчаннях. Однак, зараз, потрібно внести кілька змін у процедуру зворотного розмноження.

Причина в тому, що наші раніше виведення зворотного розмноження було для мереж з повністю зв’язаними рівнями. На щастя, просто змінити виведення для згорткових шарів та шарів, що об'єднують максимум.

Основними рівняннями зворотного розповсюдження в мережі з повністю зв’язаними шарами є (*BP*1)-(*BP*4). Припустимо, є мережа, що містить згортковий рівень, рівень максимального об’єднання та повністю підключений вихідний рівень, як у мережі, що було показано раніше. Необхідно відслідковувати, як змінюються рівняння зворотного розповсюдження при даному підході.

2.5. Висновки до розділу

В даному розділі було розглянуто питання цифрової обробки зображень, як окремої галузі математичної статистики, штучного інтелекту (теорії нейронних мереж). Визначено, що методи цифрової обробки зображень, зазвичай, здійснюють перетворення, очищення та трансформацію зображення в цифровий формат даних і інші уявлення, які можуть безпосередньо оброблятися системою розпізнавання рукописного тексту.

Варто розділити проблему розпізнавання рукописних цифр на дві підзадачі:

1) розбити зображення, що містить багато цифр, на послідовність окремих зображень, кожна з яких містить одну цифру;

2) розпізнавання окремих цифр.

При розпізнаванні рукописного тексту є ряд труднощів з зображеннями, тому що будови можуть бути нерівними, букви з різних рядків перетинатися з іншими, а відстань між рядками змінюватись в широкому діапозоні. Алгоритм сегментації строк повинен аналізувати випадкові перетини траєкторій сусідніх рядків.

Після того, як рядки побудовані, перед нами постає наступне завдання - як правильним чином розділити їх на слова. Це, як правило, нескладно для

друкованого тексту, де відносно чітко визначена різниця у відстані між буквами і між словами. У разі рукописного тексту в багатьох випадках не можна просто по відстані між буквами визначити, зіткнулися ми з між словами або у середині слова.

Одним із підходів є випробування багатьох різних способів сегментування зображення за допомогою індивідуального класифікатора цифр для оцінки кожної пробної сегментації. Пробна сегментація отримує високий бал, якщо окремий класифікатор цифр впевнений у своїй класифікації у всіх сегментах, і низький бал, якщо класифікатор має багато проблем в одному або декількох сегментах.

Для розпізнавання окремих цифр необхідно використовувати тришарову нейронну мережу.

РОЗДІЛ 3
ПРОЕКТУВАННЯ МОДУЛЮ РОЗПІЗНАВАННЯ РУКОПИСНОГО ТЕКСТУ

Центральним елементом є мережевий клас, який використовуємо для представлення нейронної мережі. Код наведено в Додатку А і використовується для ініціалізації мережевого об’єкта.

В розділі 2 було розглнуто, як утоврюються згорткові нейронні мережі. Розглянемо, як вони працюють на практиці, впроваджуючи деякі згорткові мережі та застосовуючи їх до проблеми класифікації цифр *MNIST*.

3.1. Описання програмного модуля розпізнавання рукописного тексту

Програма, яку використовуватимемо для цього, називається *network_text.py*, реалізована за допомогою *Python* та матричної бібліотеки *Numpy*. Ця програма заглибилась у деталі зворотного розповсюдження, стохастичного градієнтного спуску. Також в програмі будемо використовувати бібліотеку машинного навчання, відому як *Theano*: *CPI* та *GPU Math Expression Compiler* у *Python*.

Theano також є основою для популярного *Pylearn*2 і Керас бібліотеки нейронних мереж. Інші популярні бібліотеки нейронних мереж на момент написання дипломної роботи включають Кава і Факел. Використання *Theano* дозволяє легко реалізувати зворотне розповсюдження для згорткових нейронних мереж, оскільки воно автоматично обчислює всі пов'язані відображення. Теано також є набагато швидшим за наш попередній код (який був написаний так, щоб його було легко зрозуміти, а не швидко), і це робить практичним навчання більш складних мереж. Зокрема, однією чудовою особливістю *Theano* є те, що він може запускати код або на центральному процесорі, або, якщо є, на графічному процесорі. Запуск на графічному процесорі забезпечує значний пришвидшення і, знову ж таки, допомагає зробити практичним навчання більш складних мереж.

Для початку роботи потрібно буде запустити *Theano* у системі. Щоб встановити *Theano*. Щоб запустити *network_text.py*, вам потрібно встановити прапор *GPU* на *True* або *False* (відповідно) у джерелі *network*3.*py*. Окрім цього, можна знайти *Theano* і працювати на графічному процесорі. У Інтернеті також є навчальні посібники, які легко знайти за допомогою *Google*, які допоможуть вам налагодити роботу.

Якщо немає графічного процесора, доступного локально, можна розглянути додаток на *Amazon Web Services* (екземпляри *EC*2 *G*2). Навіть із графічним процесором для виконання коду знадобиться деякий час. Багато експериментів тривають від хвилин до годин. На центральному процесорі можуть знадобитися дні, щоб провести найскладніші експерименти. Якщо використовувати центральний процесор, можна зменшити кількість навчальних епох для більш складних експериментів або, можливо, повністю їх опустити.

Щоб отримати базову лінію, почнемо з неглибокої архітектури, використовуючи лише один прихований шар, що містить 100 прихованих нейронів. Будемо тренуватися для 60 епох, використовуючи рівень навчання η = 0,1, розмір міні-партії 10, і відсутність регуляризації (рис. 3.1).

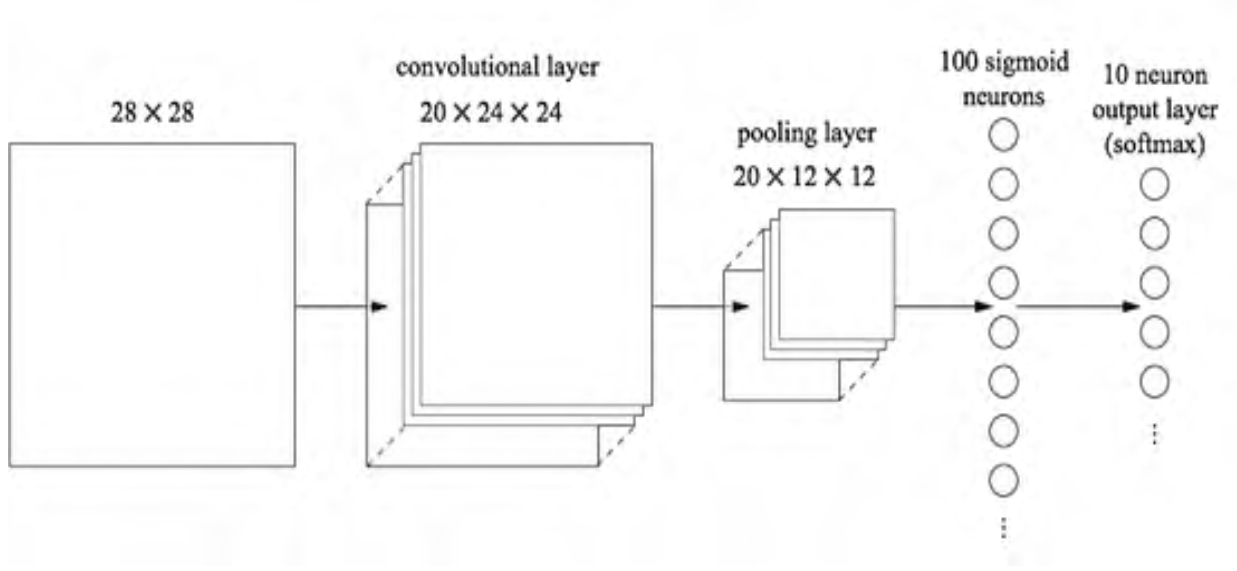

Рис. 3.1. Налаштування нейронної мережі

Вставимо згортковий шар, безпосередньо на початку мережі. Використаємо 5 на 5 місцевих рецептивних поля, довжина кроку 1, і 20 зображень особливості карт. Також вставимо шар максимального об’єднання, який поєднує в собі

використовувані функції 2 на 2 об'єднання вікон. Тож загальна архітектура мережі схожа на архітектуру, обговорену в останньому розділі, але з додатковим повністю зв’язаним шаром.

У цій архітектурі можемо використовувати згортковий та об’єднуючий шари як про вивчення місцевої просторової структури у вхідному навчальному зображенні, тоді як пізніший, повністю зв’язаний шар навчається на більш абстрактному рівні, інтегруючи глобальну інформацію з усього зображення. Це загальна закономірність у згорткових нейронних мережах.

Навчимо таку мережу і подивимось, як вона працює, продовжувючи використовувати міні-партію розміром 10. Насправді, можна прискорити навчання, використовуючи більші міні-партії. Але в роботі продовжуємо використовувати той самий розмір міні-партії, здебільшого для узгодження з експериментами в попередніх весріях програми:

```
>>> expanded_traini_d, _, _ = network_text.load_data_shared(
    "../data/mnist_expanded_text.pkl.gz")
>>> net = Network([
    ConvPoolLayer(image_shape=(mini_batch_size_text, 1, 28, 28),
                  filter_shape_text =(20, 1, 5, 5),
                  poolsize_text =(2, 2),
                  activation_fn_text =ReLU),
    ConvPoolLayer(image_shape_text =(mini_batch_size, 20, 12, 12),
                  filter_shape_text =(40, 20, 5, 5),
                  poolsize_text =(2, 2),
                  activation_fn_text =ReLU),
    FullyConnectedLayer(n_in_text =40*4*4, n_out_text =100,
activation_fn_text =ReLU),
    SoftmaxLayer(n_in_text =100, n_out_text =10)], mini_batch_size_text)
>>> net.SGD(expanded_training_data_text, 60, mini_batch_size_text, 0.03,
        validation_data_text, test_data_text, lmbda_text =0.1)
```

Це дозволяє отримати 98,78 відсотків точності, що є значним покращенням порівняно з будь-яким із попередніх результатів. Дійсно, зменшили рівень помилок краще ніж на третину, що є значним покращенням.

Визначаючи структуру мережі, розглядались згорткові та об'єднуючі шари як один шар. Чи розглядаються вони як окремі шари чи як один шар, це певною мірою питання смаку. Але в програмі *network_text.py* розглядаються як один шар, оскільки це робить код для *network_text.py* трохи компактнішим. Однак змінити *network_text.py* легко, тому шари можна вказати окремо, якщо потрібно.

Першим етапом для дослідження нейронної мережі був збір навчальної вибірки. Для цього було зібрано множину зображень намальованих літер української рукописного тексту.

Для запису літер була розроблена окрема програма із простим та зручним інтерфейсом.

Програма полегшує процес малювання літер, всі рисунки з літерами зберігаються у форматі *PNG*, а розмір кожного 260 × 360 пікселів.

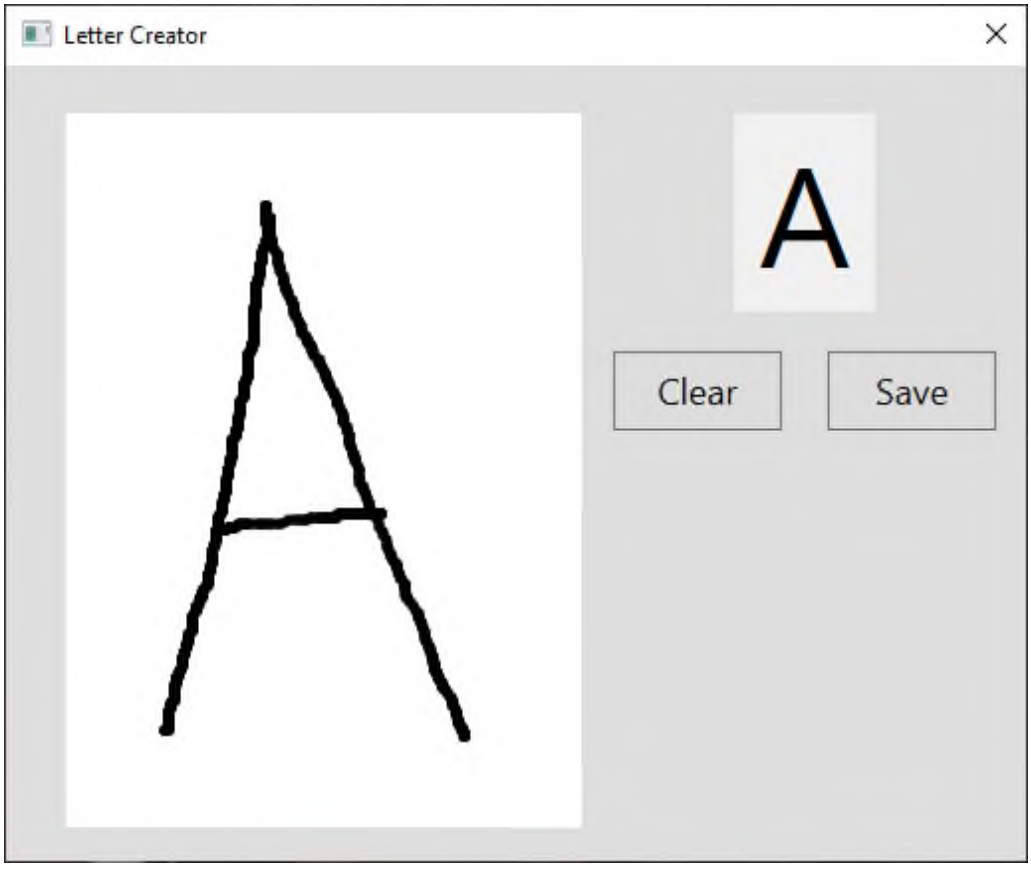

Рис. 3.3. Вікно розробленої програми розпізнавання рукописного тексту

Я отримав найкращу точність класифікації 97,8097,80відсотків. Це точність класифікації на test_data, оцінена в епоху навчання, де ми отримуємо найкращу точність класифікації на validation_data. Використання даних перевірки для прийняття рішення про те, коли оцінювати точність тесту, допомагає уникнути переобладнання даних тесту (див. Цепопереднє обговореннявикористання даних перевірки). Ми дотримуватимемось цієї практики нижче. Ваші результати можуть дещо відрізнятися, оскільки ваги та упередження мережі випадково ініціалізуються Насправді, у цьому експерименті я насправді провів три окремі прогони, навчаючи мережу з цією архітектурою. Потім я повідомив про точність тесту, яка відповідала найкращій точності перевірки з будь-якого з трьох циклів. Використання декількох прогонів допомагає зменшити різницю в результатах, що корисно при порівнянні багатьох архітектур, як ми це робимо. Я дотримувався цієї процедури нижче, за винятком випадків, коли зазначено. На практиці це мало мало значення для отриманих результатів..

Це 97,8097,80 відсоткова точність близька до 98,0498,04 відсоткова точність, отримана ще в Розділ 3, використовуючи подібну мережеву архітектуру та вивчаючи гіперпараметри. Зокрема, в обох прикладах використана неглибока мережа, що містить один прихований шар100100приховані нейрони. Обидва також тренувались для6060 епох, використовували міні-партію розміром 1010, і рівень навчання $\eta = 0,1$.

Однак існували дві відмінності в попередній мережі. По-перше, мирегуляризованийпопередня мережа, щоб допомогти зменшити наслідки переобладнання. Регуляризація поточної мережі справді покращує точність, але виграш лише невеликий, і тому ми перестанемо турбуватися про регуляризацію до пізніше. По-друге, тоді як остаточний рівень у попередній мережі використовував активації сигмоїдів та функцію перехресної ентропії, поточна мережа використовує остаточний рівень softmax та функцію витрат на вірогідність журналу. Якпоясниву розділі 3 це не велика зміна. Я не робив цього перемикання з якоїсь особливо глибокої причини - здебільшого, я це зробив, оскільки softmax плюс вартість вірогідності журналу частіше зустрічається в сучасних мережах класифікації зображень. (рис. 3.4).

Рис. 3.4. Вікно розробленої програми розпізнавання рукописного тексту з наступною літерою

На даний момент є два природні запитання. Перше запитання: що це взагалі означає застосовувати другий шар згорткового об’єднання? Насправді, ви можете думати про другий шар згорткового об'єднання як про вхідний12 × 1212 × 12"зображення", "пікселі" яких відображають наявність (або відсутність) певних локалізованих елементів у вихідному вхідному зображенні. Отже, ви можете думати про цей шар як про вхідну версію вихідного вхідного зображення. Ця версія абстрагована та ущільнена, але все ще має багато просторової структури, і тому має сенс використовувати другий шар згортання.

Це задовільна точка зору, але породжує друге питання. Вихід з попереднього шару передбачає2020 рік окремі карти функцій, і тому є 20 × 12 × 1220 × 12 × 12входи до другого рівня згортання. Це ніби у нас є2020 рікокремі зображення, що вводяться на шар згорткового об'єднання, а не одне зображення, як це було для першого шару згортання. Як повинні реагувати нейрони другого шару, що об'єднує згортки, на ці кілька вхідних зображень? Насправді ми дозволимо кожному нейрону цього шару вчитися у всіх20 × 5 × 520 × 5 × 5вхідні нейрони в місцевому рецептивному полі. Більш неофіційно: детектори функцій у

другому шарі, що об’єднує згортки, мають доступ до всіх функцій попереднього шару, але лише в межах їх конкретного місцевого рецептивного поля ** Ця проблема виникла б у першому шарі, якби вхідні зображення були кольоровими. У цьому випадку ми мали б 3 функції введення для кожного пікселя, що відповідають червоному, зеленому та синьому каналам на вхідному зображенні. Отже, ми дозволили б детекторам функцій мати доступ до всієї кольорової інформації, але лише в межах певного місцевого рецептивного поля..

Рис. 3.5. Рисунки в папці (літера «A»)

Для навчальної вибірки було намальовано кожну літеру по 144 рази, загальна кількість складає 4608 рисунків, загальний об’єм на диску складає 37,44 *Мб*. Таку велику вибірку було створено для того, щоб навчання нейронної мережі було більш точним і у майбутньому нейронна мережа робила набагато менше помилок при розпізнаванні тексту.

Мережа, яку розробили на даний момент, насправді є варіантом однієї з використаних мереж. Існує багато відмінностей у деталях, але загалом наша мережа досить схожа на описані мережі. Це хороша основа для подальших експериментів, а також для формування розуміння та інтуїції. Зокрема, існує багато способів змінити мережу, намагаючись покращити наші результати.

Для покразення роботи програми змінимо наші нейрони, щоб замість того, щоб використовувати функцію активації сигмоподібної функції,

використовували випрямлені лінійні одиниці. Тобто будемо використовувати функцію активації $f(z) \equiv max(0, z)$. Будемо тренуватися для 60 епох, із швидкістю навчання η = 0,03.

Також виявлено, що покращує результат використання деяких $l2$ регуляризацій, з параметром регуляризації λ = 0,1:

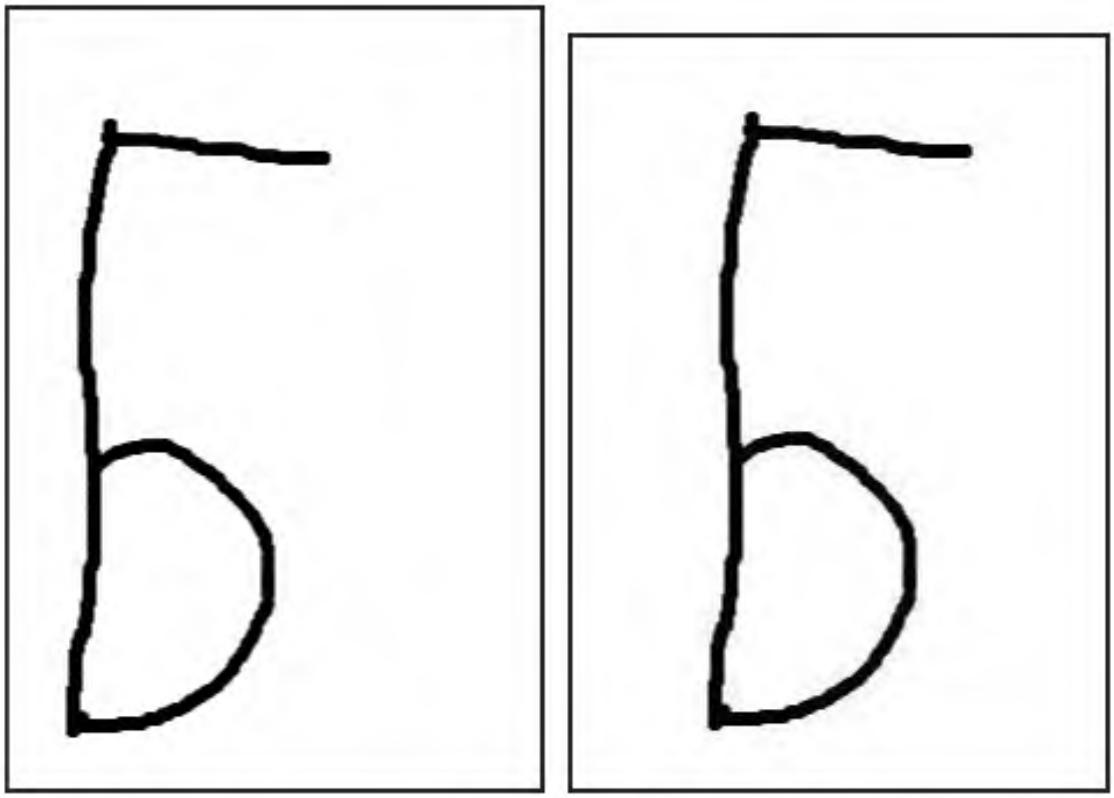

Рис. 3.6. Літера до та після центрації

3.2. Приктичне навчання НМ

в останній розділщо існують основні перешкоди для навчання в глибоких багатошарових нейронних мережах. Зокрема, ми побачили, що градієнт, як правило, досить нестабільний: коли ми переходимо від вихідного шару до попередніх шарів, градієнт має тенденцію або зникати (проблема зникаючого градієнта), або вибухати (проблема вибуху градієнта). Оскільки градієнт - це сигнал, який ми використовуємо для тренування, це створює проблеми.

Як ми уникнули цих результатів? Звичайно, відповідь така: ми не уникали цих результатів. Натомість ми зробили кілька речей, які все одно допомагають

нам продовжувати. Зокрема: (1) Використання згорткових шарів значно зменшує кількість параметрів у цих шарах, що значно полегшує навчальну проблему; (2) Використання більш потужних методів регуляризації (зокрема, відсіву та згорткових шарів) для зменшення переобладнання, що в іншому випадку є більшою проблемою в більш складних мережах; (3) Використання випрямлених лінійних одиниць замість сигмоподібних нейронів, щоб пришвидшити навчання - емпірично, часто в рази33-55; (4) Використання графічних процесорів та готовність тренуватися протягом тривалого періоду часу. Зокрема, у наших заключних експериментах, для яких ми тренувалися4040 епохи з використанням набору даних 55в рази більше, ніж вихідні дані MNIST. Раніше в книзі ми в основному тренувались3030епохи, використовуючи лише необроблені навчальні дані. Поєднуючи фактори (3) та (4), наче ми навчили якийсь фактор3030 в рази довше, ніж раніше.

Ваша відповідь може бути: "Це все? Це все, що нам потрібно було зробити, щоб навчити глибокі мережі? У чому вся суєта?"

Звичайно, ми використовували й інші ідеї: використання достатньо великих наборів даних (щоб уникнути переобладнання); використовуючи правильну функцію витрат (доуникати уповільнення навчання); використанняхороша вагова ініціалізація (також уникнути уповільнення навчання через насичення нейронів); алгоритмічне розширення навчальних даних. Ми обговорювали ці та інші ідеї в попередніх розділах і, здебільшого, мали змогу повторно використовувати ці ідеї, мало коментарів у цій главі.

З огляду на це, насправді це досить простий набір ідей. Простий, але потужний, коли використовується спільно. Почати з глибокого навчання виявилося досить легко!

Наскільки глибокі ці мережі, взагалі? Наша фінальна архітектура враховує шари, що об’єднують згортки, як одношарові 44приховані шари. Чи справді така мережа заслуговує на те, щоб її називали глибинною мережею? Звичайно,44прихованих шарів набагато більше, ніж у неглибоких мережах, які ми вивчали раніше. Більшість із цих мереж мали лише один прихований шар або іноді22приховані шари. З іншого боку, станом на 2015 рік ультрасучасні

глибинні мережі іноді мають десятки прихованих шарів. Іноді я чув, як люди приймають глибше ставлення, ніж ти, вважаючи, що якщо ти не встигаєш за Джонсом за кількістю прихованих шарів, то ти насправді не глибоко навчаєшся. Я не співчуваю такому ставленню, зокрема тому, що воно перетворює визначення глибокого навчання на те, що залежить від результату моменту. Справжнім проривом у глибокому навчанні було усвідомлення того, що практично виходити за межі неглибокого11- і 22- приховані шарові мережі, які домінували у роботі до середини 2000-х. Це справді стало значним проривом, який відкрив дослідження набагато виразніших моделей. Але крім того, кількість шарів не становить першочергового фундаментального інтересу. Швидше, використання більш глибоких мереж - це інструмент, який допомагає досягти інших цілей - наприклад, кращої точності класифікації.

Слово про процедуру:У цьому розділі ми плавно перейшли від одинарних прихованих шарів неглибоких мереж до багатошарових згорткових мереж. Все здавалося так легко! Ми вносимо зміни і, здебільшого, отримуємо вдосконалення. Якщо ви почнете експериментувати, я можу гарантувати, що не завжди все буде так гладко. Причина полягає в тому, що я представив очищений розповідь, опустивши багато експериментів - у тому числі багато невдалих експериментів. Сподіваємось, цей розчищений розповідь допоможе вам зрозуміти основні ідеї. Але це також ризикує створити неповне враження. Отримати хорошу, працюючу мережу може включати багато спроб і помилок, а іноді і розчарування. На практиці слід розраховувати на чимало експериментів. Щоб пришвидшити цей процес, вам може бути корисним переглянути обговорення розділу 3як вибрати гіперпараметри нейронної мережі, а можливо, також поглянути на деякі подальші читання, запропоновані в цьому розділі.

Код наших згорткових мереж

Добре, давайте подивимось на код нашої програми, network3.py. Структурно це схоже на програму network2.py, яку ми розробилиРозділ 3, хоча деталі різняться, завдяки використанню Theano. Ми почнемо з розгляду класу FullyConnectedLayer, який подібний до шарів, вивчених раніше в книзі. Ось код (обговорення нижче)

```
  32/4565 [..............................] - ETA: 2s - loss: 0.2300 - acc: 0.9375
 128/4565 [..............................] - ETA: 2s - loss: 0.4073 - acc: 0.8516
 224/4565 [>.............................] - ETA: 2s - loss: 0.3887 - acc: 0.8839
 288/4565 [>.............................] - ETA: 3s - loss: 0.3693 - acc: 0.9028
 384/4565 [=>............................] - ETA: 2s - loss: 0.3839 - acc: 0.8906
 480/4565 [==>...........................] - ETA: 2s - loss: 0.3967 - acc: 0.8875
 576/4565 [==>...........................] - ETA: 2s - loss: 0.4014 - acc: 0.8854
 672/4565 [===>..........................] - ETA: 2s - loss: 0.3941 - acc: 0.8869
 768/4565 [====>.........................] - ETA: 2s - loss: 0.3863 - acc: 0.8945
 864/4565 [====>.........................] - ETA: 2s - loss: 0.3781 - acc: 0.8947
 960/4565 [=====>........................] - ETA: 2s - loss: 0.3921 - acc: 0.8875
1056/4565 [=====>........................] - ETA: 2s - loss: 0.3882 - acc: 0.8902
1152/4565 [======>.......................] - ETA: 2s - loss: 0.3908 - acc: 0.8898
1248/4565 [=======>......................] - ETA: 2s - loss: 0.3941 - acc: 0.8878
1344/4565 [=======>......................] - ETA: 2s - loss: 0.3928 - acc: 0.8876
1440/4565 [========>.....................] - ETA: 2s - loss: 0.3841 - acc: 0.8910
1536/4565 [=========>....................] - ETA: 2s - loss: 0.3821 - acc: 0.8932
1632/4565 [=========>....................] - ETA: 1s - loss: 0.3840 - acc: 0.8915
1728/4565 [==========>...................] - ETA: 1s - loss: 0.3775 - acc: 0.8941
1824/4565 [==========>...................] - ETA: 1s - loss: 0.3723 - acc: 0.8953
```

Рис. 3.7. Процес навчання нейронної мережі

Ще один спосіб, на який можемо сподіватися покращити наші результати, – це алгоритмічне розширення навчальних даних. Простий спосіб розширення навчальних даних – це зміщення кожного навчального зображення на один піксель, або на один піксель, або на один піксель, вниз, наліво на один піксель. Можемо зробити це, запустивши програму *expand_mnist.py* із підказки оболонки *python expand_mnist.py*.

Запуск цієї програми отримує доступ до 50 000 *MNIST* навчальних зображень, і готує розширений навчальний набір, з 250 000 навчальними зображеннями. Потім можемо використовувати ці навчальні зображення для навчання нашої мережі. Будемо використовувати ту саму мережу, що і вище, з випрямленими лінійними одиницями. У початкових експериментах було зменшено кількість епох тренувань – це мало сенс, оскільки тренуємося з ними в 55 разів більше даних. Але насправді розширення даних виявилось значно зменшити ефект переобладнання. І ось, після деяких експериментів, врешті-решт повернулись до тренувань 60 епох.

Використовуючи розширені навчальні дані, отримано 99,37 відсоток точності навчання. Тож ця зміна суттєво покращує точність класифікації.

У цьому коді розмір списку містить кількість нейронів у відповідних шарах. Так, наприклад, якщо хочемо створити мережевий об’єкт з 2 нейронами в першому шарі, 3 нейронами у другому шарі та 1 нейроном у кінцевому шарі,

Всі упередження та ваги в об'єкті Мережі ініціалізуються випадковим чином, використовуючи функцію *Numpy np.random.randn* для генерації гауссових розподілів із середнім значенням 0 і стандартне відхилення 1. Ця випадкова ініціалізація дає нашому алгоритму стохастичного градієнтного спуску місце, з якого слід почати. Необхідно звернути увагу, що код ініціалізації мережі передбачає, що перший рівень нейронів є вхідним шаром, і не дозволяє встановлювати будь-які упередження для цих нейронів, оскільки упередження завжди використовуються лише для обчислення результатів з наступних шарів.

Також необхідно зауважити, що зміщення та ваги зберігаються як списки матриць *Numpy*. Так, наприклад, *net.weights* [10] – матриця Нумпі, що зберігає ваги, що з'єднують другий і третій шари нейронів. Це така матриця w_{jk} – вага для зв'язку між *k*-им нейроном у другому шарі і *j*-им нейроном у третьому шарі. Це впорядкування *j* і *k* індекси можуть здатися дивними – безумовно, було б більше сенсу поміняти *j* і *k* індекси навколо.

Дані навчання – це список кортежів (*x, y*), що представляють вхідні дані для навчання та відповідні бажані результати. Змінні *epochs* і *mini_batch_size* – кількість епох, для яких потрібно тренуватися, і розмір міні-партій, які слід використовувати під час вибірки. *eta* – швидкість навчання,η. Якщо необхідно навести необов’язковий аргумент *test_data*, програма найвірогідніше буде оцінювати мережу після кожної епохи навчання та друкувати частковий прогрес, якого вона досягла. Це корисно для відстеження прогресу, але істотно сповільнює ситуацію.

Код працює наступним чином. У кожну епоху він починається з випадкового перемішування навчальних даних, а потім розділяє їх на міні-партії відповідного розміру. Це простий спосіб вибіркової вибірки з навчальних даних. Потім для кожного *mini_batch* застосовуємо один крок градієнтного спуску. Це робиться за допомогою коду *self.update_mini_batch* (*mini_batch, eta*), який

оновлює ваги та упередження мережі відповідно до однієї ітерації градієнтного спуску, використовуючи лише навчальні дані в *mini_batch*.

Це викликає алгоритм зворотного розповсюдження, який є швидким способом обчислення градієнта функції витрат. Отже *update_mini_batch* працює досить просто, обчислюючи ці градієнти для кожного з навчальних прикладів в *mini_batch*, а після цього належним чином проводить оновлення для *self.weights* та *self.biases*.

Давайте розглянемо повну програму, включаючи рядки документації, які представлено у Додатку А. Окрім *self.backprop*, програма сама собою пояснюється – усі важкі дії піднімаються в *self.SGD* та *self.update_mini_batch*, про які вже говорили. Метод *self.backprop* використовує кілька додаткових функцій для обчислення градієнта, а саме *sigmoid_prime*, який обчислює похідну від *σ function* та *self.cost_derivative*.

Можна отримати основну ідею деталей даної програми, просто переглянувши рядок коду та документації. Хоча програма здається тривіальною, більша частина коду – це рядки документації, призначені для полегшення розуміння коду. Вданому випадку програма містить лише 74 рядки коду без пробілів і коментарів.

Тобто навчена мережа дає коефіцієнт класифікації приблизно 6595 відсотків – 95.42 відсотків на піку ("Епоха 28"). Це дуже обнадійливо з першої спроби. Якщо запускаємо код, то результати не обов'язково будуть такими ж, як представлено, оскільки ініціалізуємо мережу, використовуючи випадкові (різні) ваги та упередження. Для отримання представлених результатів зроблено вибір найкращих із трьох запусків.

Давайте повторимо вищезазначений експеримент, змінивши кількість прихованих нейронів на 100100. Як це було раніше, якщо запускаємо код, слід попередити, що його виконання займає досить багато часу (на моємо компьютері цей експеримент займає десятки секунд для кожної навчальної епохи).

Звичайно, це покращує результати до 96,59 відсотків. Принаймні в цьому випадку використання більш прихованих нейронів допомагає отримати кращі результати. Тести вказують на певні відмінності в результатах цього

експерименту, а деякі тренувальні прогони дають результати набагато гірші. Використання методів, представлених у розділі 2, значно зменшить різницю у продуктивності в різних навчальних пробігах для наших мереж.

Звичайно, щоб отримати ці точності, довелося зробити конкретний вибір щодо кількості епох навчання, розміру міні-партії та швидкості навчання, η. Як вже згадувано вище, вони відомі як гіперпараметри для даної нейронної мережі, щоб відрізнити їх від параметрів (ваг та упереджень), засвоєних цим алгоритмом навчання. Якщо досить погано обрати гіперпараметри, то можливо отримати досить погані результати.

Спробуємо надзвичайно просту ідею: розглянемо, наскільки темним є зображення. Наприклад, зображення 2 зазвичай буде трохи темнішим, ніж зображення 1, лише тому, що більше пікселів почорніло, як ілюструють наступні приклади:

Це пропонує використовувати навчальні дані для обчислення середньої темряви для кожної цифри, 0,1,2,…,9. Коли представляють нове зображення, обчислюємо, наскільки темним воно є, і тоді здогадуємось, що це будь-яка цифра має найближчу середню темряву. Це проста процедура, і її легко кодувати, тому не будемо явно виписувати код. Але це значне покращення порівняно з випадковим здогадуванням, отриманням 2 225 з 10 000 тестові зображення правильні, тобто 22.25 відсоткова точність.

Досить легко знайти інші варіанти, які можуть досягати точності від 20 до 50 в відсотковому діапазоні. Якщо трохи попрацювати та потоестувати системо, то можна дійти до 50 відсотків. Але для отримання набагато більшої точності необхідно почати використання встановлених алгоритмів машинного навчання. Спробуємо використати один із найвідоміших алгоритмів, машину з підтримкою векторів або *SVM*. Необхідно використовувати бібліотеку *Python* з назвою *scikit-learn*, яка забезпечує простий інтерфейс *Python* для швидкої бібліотеки на базі *C* для *SVM*, відомих як ЛІБСВМ.

Якщо запустимо класифікатор *SVM scikit-learn* за допомогою налаштувань за замовчуванням, тоді він отримає 9 435 з 10 000 тестових зображень правильними. Це досить значне покращення цього підходу до класифікації

зображення на основі того, наскільки воно темне. В даному випадку, це означає, що *SVM* працює приблизно так само добре, як і нейронні мережі описані вище, але в деяких варіаціях може бути трохи гірше. Далі будуть представлені методи, які нададуть змогу досить суттєво вдосконалити дану нейронну мережу, щоб вона працювала набагато краще, ніж *SVM*.

Однак на цьому історія ще не закінчена. Результат 9 435 з 10 000 – це налаштування *scikit-learn* за замовчуванням для *SVM*. *SVM* мають ряд налаштовуваних параметрів, і можна шукати параметри, які покращують цю нестандартну продуктивність.

Більша частина методу *__init__* є зрозумілою, але кілька нотатків в документації можуть допомогти пояснити код. Як зазвичай, випадково ініціалізуємо ваги та упередження як звичайні випадкові величини з відповідними стандартними відхиленнями. Рядки, що роблять це, виглядають трохи забороняючими. Однак більшість ускладнень полягає лише у завантаженні ваг та упереджень у те, що Теано називає спільними змінними. Це гарантує, що ці змінні можуть бути оброблені на графічному процесорі, якщо такий присутній в системі.

Необхідно зауважити, що ця ініціалізація ваги та упередженості призначена для функції активації сигмоїдної форми, яка була обговорена раніше. В ідеалі потрібно дещо по-іншому ініціалізувати ваги та упередження для таких функцій активації, як *tanh* та випрямлена лінійна функція. Це допомагає покращити процес навчання та вирішити проблеми минулих методів. Метод *__init_text __* закінчується *self_text.params = [self_text.w, self_text.b]*. Це зручний спосіб об’єднати всі вивчувані параметри, пов’язані з шаром. Пізніше метод *Network.SGD* використовуватиме атрибути *params*, щоб з'ясувати, яким змінним у мережевому екземплярі можна навчитися.

Метод *set_inpt* використовується для встановлення вхідного сигналу на рівень і для обчислення відповідного виводу. Використовуємо ім'я *inpt*, а не *input*, оскільки введення – це вбудована функція в *Python*, і робота з вбудованими процесами спричиняє непередбачувану поведінку та важкі для діагностики помилки. Необхідно звернути увагу, що фактично встановлюємо введення двома

окремими способами: як *self.inpt* та *self.inpt_dropout*. Це робиться тому, що під час навчання можливо використати відсів. Якщо це так, тоді можливо видалити частку *self_text.p_dropout* нейронів. Ось що робить функція *dropout_layer* у другому останньому рядку методу *set_inpt*. Отже, *self.inpt_dropout* та *self.output_dropout* використовуються під час навчання, тоді як *self.inpt* і *self.output* використовуються для всіх інших цілей, наприклад, для оцінки точності даних перевірки та тестування.

Визначення класів *ConvPoolLayer* та *SoftmaxLayer* подібні до *FullyConnectedLayer*. Дійсно, вони настільки близькі, що не будемо описувати тут уривок коду, який представлено у Додатку А.

Однак варто зазначити пару незначних відмінностей у деталях. Найбільш очевидно, що і в *ConvPoolLayer*, і в *SoftmaxLayer* обчислює вихідні активації способом, відповідним цьому типу шару. На щастя, *Theano* робить це простим, забезпечуючи вбудовані операції для обчислення згорток, макс-пулінгу та функції *softmax*.

Менш очевидно, коли було введено шар *softmax*, ніколи не обговорювали, як ініціалізувати ваги та упередження. В іншому стверджували, що для сигмоподібних шарів повинні ініціалізувати ваги, використовуючи належним чином параметризовані нормальні випадкові величини. Але цей евристичний аргумент був специфічним для сигмоїдних нейронів (і, з деякими поправками, для нейронів Тан). Однак немає особливої причини, за якою аргумент повинен застосовуватися до шарів *softmax*. Тож немає апріорних причин застосовувати цю ініціалізацію знову. Замість цього ініціалізуємо всі ваги та упередження 0. Це досить спеціальна процедура, але на практиці працює досить добре.

3.3. Експериментальне розпізнавання символів

Тобто навчена мережа дає нам коефіцієнт класифікації приблизно 9595 відсотків - 95.4295.42відсотків на піку ("Епоха 28")! Це дуже обнадійливо з першої спроби. Однак я повинен попередити вас, що якщо ви запускаєте код, то ваші результати не обов'язково будуть такими ж, як і у мене, оскільки ми

ініціалізуємо нашу мережу, використовуючи (різні) випадкові ваги та упередження. Для отримання результатів у цій главі я зробив найкращий із трьох запусків.

Давайте повторимо вищезазначений експеримент, змінивши кількість прихованих нейронів на 100100. Як це було раніше, якщо ви запускаєте код під час читання, вам слід попередити, що його виконання займає досить багато часу (на моїй машині цей експеримент займає десятки секунд для кожної навчальної епохи), тому розумно продовжуйте читати паралельно під час виконання коду.

>>> чистий = мережі.Мережа ([784, 100, 10]))

>>> чистий.SGD (навчальні_дані, 30, 10, 3.0, дані_тестів=test_data)

Звичайно, це покращує результати 96,5996,59відсотків. Принаймні в цьому випадку використання більш прихованих нейронів допомагає нам отримати кращі результати ** Відгуки читачів вказують на певні відмінності в результатах цього експерименту, а деякі тренувальні прогони дають результати набагато гірші. Використання методів, представлених у главі 3, значно зменшить різницю у продуктивності в різних навчальних пробігах для наших мереж..

Звичайно, щоб отримати ці точності, мені довелося зробити конкретний вибір щодо кількості епох навчання, розміру міні-партії та швидкості навчання, ηη. Як я вже згадував вище, вони відомі як гіперпараметри для нашої нейронної мережі, щоб відрізнити їх від параметрів (ваг та упереджень), засвоєних нашим алгоритмом навчання. Якщо ми погано вибираємо наші гіперпараметри, ми можемо отримати погані результати. Припустимо, наприклад, що ми вибрали таку швидкість навчанняη = 0,001η = 0,001,

>>> чистий = мережі.Мережа ([784, 100, 10]))

>>> чистий.SGD (навчальні_дані, 30, 10, 0,001, дані_тестів=test_data)

Результати набагато менш обнадійливі,

Епоха 0: 1139/10000

Епоха 1: 1136/10000

Епоха 2: 1135/10000

...

Епоха 27: 2101/10000

Епоха 28: 2123/10000

Епоха 29: 2142/10000

Однак ви бачите, що продуктивність мережі з часом поступово покращується. Це передбачає збільшення рівня навчання, скажімоη = 0,01η = 0,01. Якщо ми це зробимо, ми отримаємо кращі результати, що передбачає повторне збільшення рівня навчання. (Якщо внесення змін покращує ситуацію, спробуйте зробити більше!) Якщо ми зробимо це кілька разів, ми в результаті отримаємо швидкість навчання приблизноη = 1,0η = 1,0 (і, можливо, тонка настройка на 3.03.0), що є близьким до наших попередніх експериментів. Отже, хоча ми спочатку неправильно вибрали гіперпараметри, ми принаймні отримали достатньо інформації, щоб допомогти нам покращити вибір гіперпараметрів.

Загалом, налагодження нейронної мережі може бути складним завданням. Це особливо вірно, коли початковий вибір гіперпараметрів дає результати не кращі за випадковий шум. Припустимо, ми спробуємо успішну архітектуру 30 прихованих нейронних мереж з попередньої, але зі зміною швидкості навчання наη = 100,0η = 100,0:

>>> чистий = мережі.Мережа ([784, 30, 10]))

>>> чистий.SGD (навчальні_дані, 30, 10, 100,0, дані_тестів=test_data)

На даний момент ми насправді зайшли занадто далеко, і рівень навчання занадто високий:

Епоха 0: 1009 / 10000

Епоха 1: 1009 / 10000

Епоха 2: 1009 / 10000

Епоха 3: 1009 / 10000

...

Епоха 27: 982 / 10000

Епоха 28: 982 / 10000

Епоха 29: 982 / 10000

А тепер уявіть, що ми вперше прийшли до цієї проблеми. Звичайно, з попередніх експериментів ми знаємо, що правильно робити, це зменшувати рівень навчання. Але якби ми вперше стикалися з цією проблемою, тоді у

вихідних даних не було б багато чого, що допомогло б нам робити, що робити. Ми можемо турбуватися не лише про рівень навчання, але і про всі інші аспекти нашої нейронної мережі. Ми можемо задатися питанням, чи не ініціалізували ми ваги та упередження таким чином, що ускладнює мережу для навчання? А може, у нас недостатньо даних про навчання, щоб отримати осмислене навчання? Можливо, ми не бігали достатньо епох? А може, нейронній мережі з такою архітектурою неможливо навчитися розпізнавати рукописні цифри? Можливо, рівень навчання занадто низький? А може, рівень навчання занадто високий? (рис. 3.7).

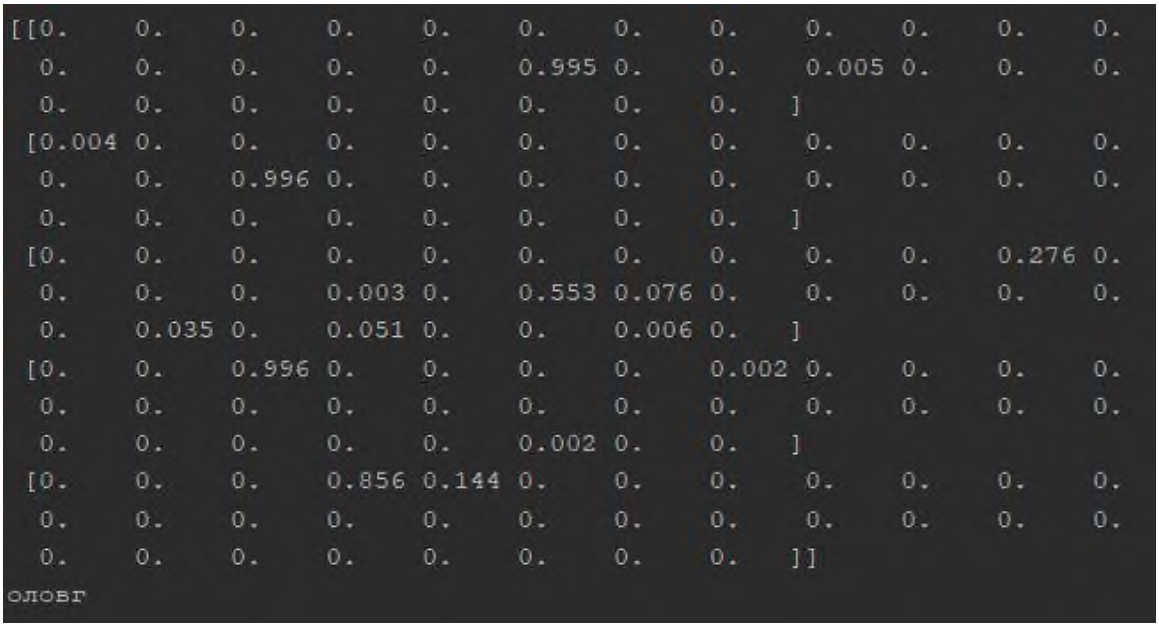

Рис 3.7. Вірогідність правильного розпізнаня

Після всіх процедур з зображенням, завантажується модель «*NETWORK.h5*», котра була навчена до цього, потім використовуючи стандартний метод «*predict*», котрий приймає в себе зображення і далі нейронна мережа аналізує зображення і отримуються данні про вірогідність тієї чи іншої літери методом «*argmax*», і програма знаходячи індекси найбільшої вірогідності в хеш таблиці алфавіту виводить літеру, яку нейронна мережа розпізнала.

3.4. Аналіз роботи основної частини

Як бачите, ці цифри насправді такі ж, як і наведені на на початку цієї главияк виклик для визнання. Звичайно, під час тестування нашої мережі ми попросимо її розпізнати зображення, яких немає в навчальному наборі!

Дані MNIST складаються з двох частин. Перша частина містить 60 000 зображень, які будуть використані як навчальні дані. Ці зображення - це відскановані зразки почерку від 250 осіб, половина з яких - співробітники Бюро перепису населення США, а половина - учні середніх шкіл. Зображення мають градації сірого та мають розмір 28 на 28 пікселів. Друга частина набору даних MNIST - це 10 000 зображень, які будуть використані як тестові дані. Знову ж таки, це 28 на 28 зображень у градаціях сірого. Ми використаємо дані тесту, щоб оцінити, наскільки добре наша нейронна мережа навчилася розпізнавати цифри. Щоб зробити це хорошим тестом успішності, дані тесту були взяті у 250 осіб, відмінних від вихідних даних про навчання (хоча все-таки група розділена між працівниками Бюро перепису населення та студентами середніх шкіл). Це допомагає нам упевнитись, що наша система може розпізнавати цифри від людей, яких це не писало.

3.5. Тестування основних компонентів

Під час тестування програми для збірки вибірки була знайдена проблема, яка критично впливала на майбутнє навчання нейронної мережі. Коли користувач запускає програму і починає малювати майбутній дата сет і зберігає файл, то рисунки не зберігались, була помилка в коді програми, був не вірно вказаний шлях до каталогу. Також була помилка, коли користувач починав малювати, полотну не вистачало частоти оновлення і замість рівної лінії були точки, у майбутньому нейронна мережа мала би не коректний процес навчання через цю помилку (рис. 3.8).

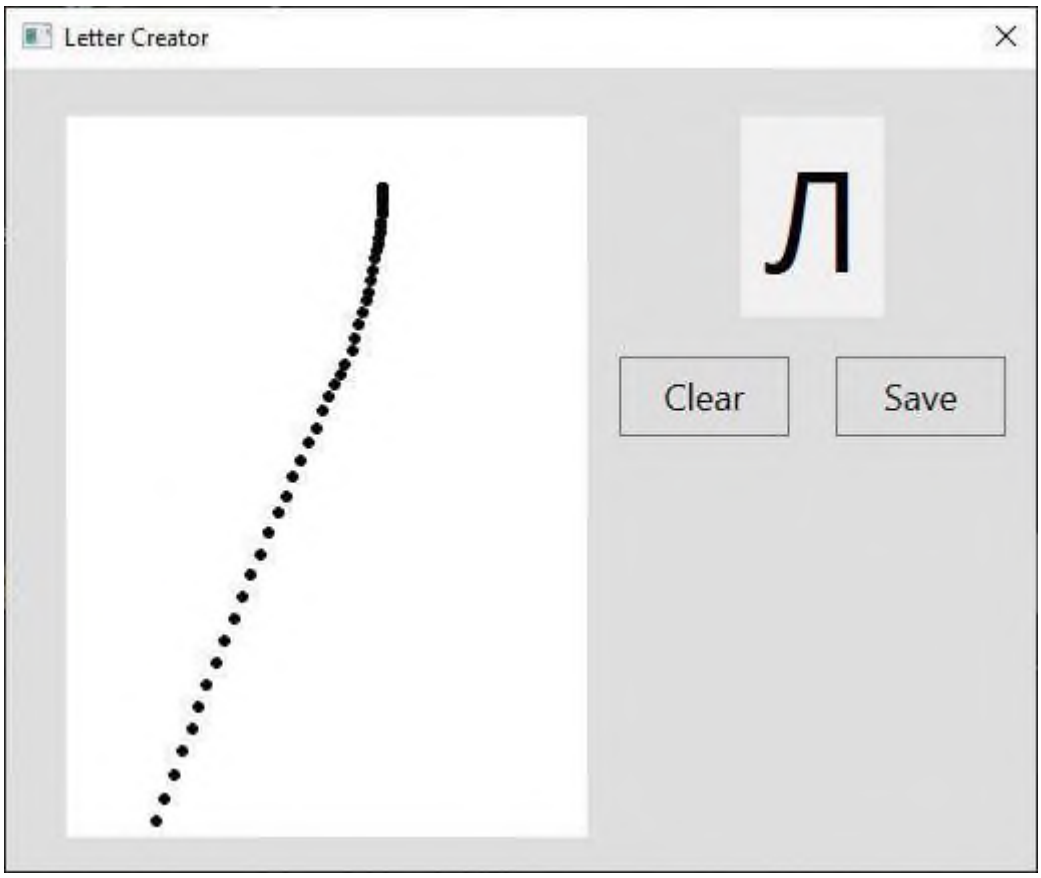

Рис. 3.8. Недостатня швидкість оновлення полотна

Точність розпізнавання – ступінь наближення результату розпізнавання до літер зображених на рисунку. Чим менша різниця між результатом і істинним значенням, тим вища точність розпізнавання. Кількісно точність розпізнавання може бути оцінена через:

– похибку розпізнавання;
– збіжність з літерами з зображення;
– невизначеність розпізнавання.

Чим більше користувач створить вибірки тим краще буде для навчання нейронної мережі, а саме, краще за все буде, якщо вибірку буде створювати не один користувач, тому що нейронна мережа матиме змогу отримати різноманітну вибірку і навчитися більш точному розпізнаванню текстів з рисунків. Користувач також повинен мати на увазі, що якщо поміж вибірки буде некоректне зображення, то нейронна мережа проведе навчання по цьому рисунку і всі подальші результати можуть бути не задовільними.

3.6. Робота з розпізнаванням

Для розпізнавання було створено файл з зображенням на ньому слова «олово».

Рис. 3.9. Слово для розпізнавання

Після старту розпізнавання був отриманий незадовільний результат з розпізнавання, нейронна мережа не змогла правильно розпізнати останню літеру зображену на рисунку, тому що остання літера була намальована з «хвостиком», саме тому нейронна мережа розпізнала літеру як, літеру «Г».

```
[[0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.
  0.    0.    0.    0.    0.    0.995 0.    0.    0.005 0.    0.    0.
  0.    0.    0.    0.    0.    0.    0.    0.   ]
 [0.004 0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.
  0.    0.    0.996 0.    0.    0.    0.    0.    0.    0.    0.    0.
  0.    0.    0.    0.    0.    0.    0.    0.   ]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.276 0.
  0.    0.    0.    0.003 0.    0.553 0.076 0.    0.    0.    0.    0.
  0.    0.035 0.    0.051 0.    0.    0.006 0.   ]
 [0.    0.    0.996 0.    0.    0.    0.    0.002 0.    0.    0.    0.
  0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.
  0.    0.    0.    0.    0.    0.002 0.    0.   ]
 [0.    0.    0.    0.856 0.144 0.    0.    0.    0.    0.    0.    0.
  0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.
  0.    0.    0.    0.    0.    0.    0.    0.   ]]
олоВг
```

Рис. 3.10. Вірогідності кожної літери

3.7. Висновки до розділу

В даному розділі було реалізовано систему розпізнавання рукописного тексту иа розглянута оцінка якості роботи програми та загальна інформація про тестування програмного забезпечення. Оцінка точності розпізнавання напряму залежить від кількості вибірки та кількості епох навчання нейронної мережі. Точність оцінюється шляхом порівняння результату та істинним значенням. Після виявлення та виправлення кожної помилки обов'язково слід повторити тести, щоб переконатися у працездатності програми.

В ході тестування програмного забезпечення було проаналізовано вихідні результати та оцінена правильність розпізнавання тексту програмою. Програма має велику точність розпізнавання, яку видно по вірогідностям розпізнавання кожної з літер.

ВИСНОВКИ

Останнім часом спостерігається розвиток нейронних мереж, а разом з ними і всебічне їх використання. Процес розпізнавання зображень є складною багатоетапною процедурою. Багатоетапність обумовлена тим, що різні завдання обробки насправді тісно зв'язані і якість вирішення однієї з них впливає на вибір методу вирішення інших. Так вибір методу розпізнавання залежить від конкретних умов представлення вхідних зображень, зокрема характеру фону, шумової обстановки і пов'язаний з вибором методів попередньої обробки, сегментації, фільтрації.

У даній дипломній роботі була розглянута тематика розбору тексту. В результаті виконання дипломної роботи мною був створений комплекс розпізнавання тексту. Крім того набув навичок роботи з професійним програмним забезпеченням даного виду, закріпив і поглибив теоретичні і практичні знання в галузі програмування

Було розглянуто поняття нейронної мережі, всі її види та характеристики.

Також було розглянуто програми, які забезпечують розпізнавання тексту. Ознайомлено з методом скелетизації, котрий при розпізнаванні символів реалізується впровадженням процесору до зняття шару за шаром з літери для покращення точності розпізнавання і не тільки, це дає змогу розпізнавати складні рисунки.

В третьому розділі було описано процес розробки програмного модулю розпізнавання рукописного тексту. Була розроблена програма для створення вибірки для нейронної мережі, та модулю по групуванню, навчанню та розпізнаванню.

Було детально розглянуто інтерфейс розробленого програмного модулю для створення вибірки. Він дуже простий та практичний, це надає програмі великі переваги: легкість керування, збору вибірки та інтуїтивний інтерфейс. Велику роль грає швидкість роботи нейронної мережі. Це дозволяє користувачу швидше проаналізувати рисунок з якого потрібно зробити розпізнавання.

Було розкрито питання, щодо застосування нейронних мереж у сферах діяльності зв’язаних з документообігом. Нейронні мережі стрімко розвиваються в безлічі інших областей, які прагнуть скоротити час опрацювання того чи іншого процесу, підвищити ефективність.

Також були наведені основні фрагменти коду програмної частини які відповідають за організацію роботи обчислень в програмі, словесно описані алгоритми роботи створеного засобу для розпізнавання тексту.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

Додаток А